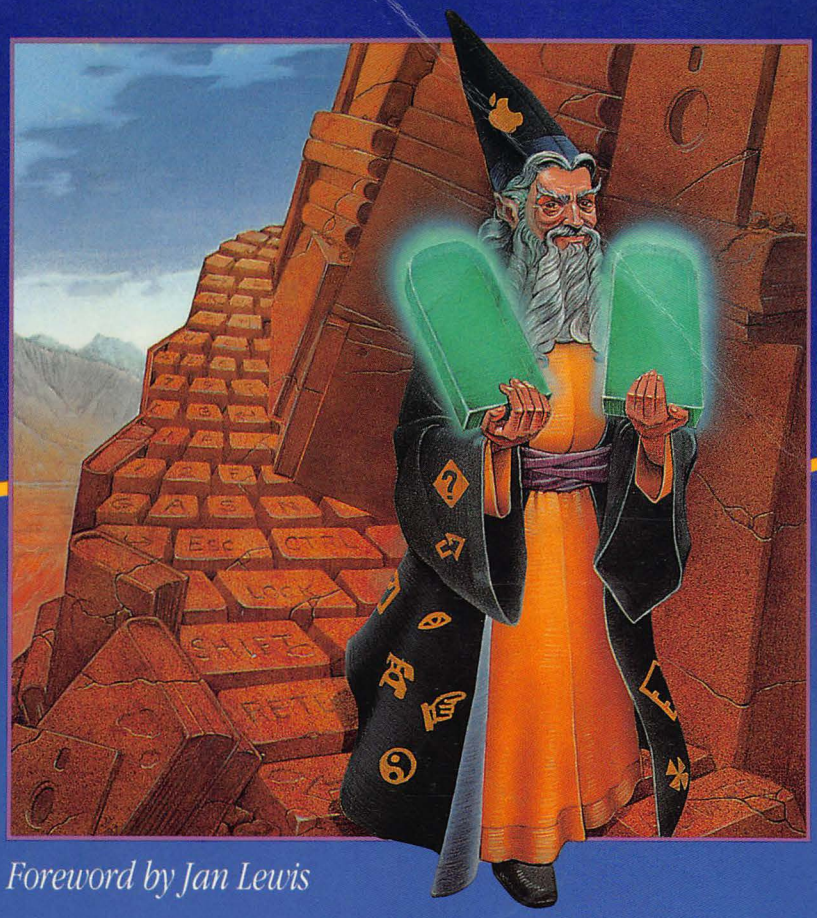
HAYDEN *Macintosh Library* BOOKS

48430

Available on Disk

The Waite Group's HyperTalk™ Bible

Covers HyperCard™ International Version 1.2.2



Foreword by Jan Lewis

Mitchell Waite
Stephen Prata
Ted Jones

HyperTalk Version 1.2.2 Language Syntax

1. Keywords

do
else
end
exit
global
if
next
on
pass
repeat
return
send
then

2. Commands

add *expr* to *expr*, *169*
arrowKey *keyName*, *262*
ask *question*, *124*
beep *count*, *37*
choose *toolName* tool, *118*
click at *location*, *243*
close file *fileName*, *598*
close printing
convert *container to format*, *388*
delete *chunk*
dial *expr*, *221*
divide *dest* by *expr*, *169*
doMenu *menuItem*, *117*
drag from *start* to *finish*, *512*
edit script of *object*, *503*
enterKey, *260*
find *expr*, *363*
find string *expr* (1.2)
find whole *expr* (1.2)
functionKey *keyNumber*, *263*
get *expr*, *149*
go *bkgndDescriptor*, *90*
go *cardDescriptor*, *90*
go stackName, *90*
help, *218*
hide background picture (1.2), *83*
hide card picture (1.2), *83*
hide menuBar, *83*
hide *part*, *83*
hide *window*, *83*
lock screen (1.2), *572*
multiply *dest by expr*,, *169*
open *application*
open file *fileName*, *594*
open printing
play stop, *482*
play *"voice"* [*tempo*] *"notes"*, *104*
pop card [*prepostion dest*], *94*
print card, *493*
print *cardDescriptor*, *493*
print *doc* with *app*
print *expr* cards
push *cardDescriptor*, *94*
put *expr* [*preposition dest*], *46*
read from file, *596*
reset paint, *526*
returnKey, *260*
select *objectDescriptor* (1.2), *371*
set *property to value*, *50*
show (cards), *83*
show background picture (1.2), *83*
show card picture (1.2), *83*
show menuBar, *588*
show number cards, *83*
show *part* [*at h,v*], *83*
show *window* [*at h,v*], *83*
sort [*dir*] [*style*] by *expr*
subtract *expr* from *dest*, *169*
tabKey, *260*
type *expr*, *264*
unlock screen [with *visEffect*] (1.2), *572*
visual *effectName*, *107*
wait *time*, *121*
wait until *condition*, *334*
wait while *condition*, *335*
write *source* to file *fileName*, *595*

3. Functions

Note: all functions have alternative format—*the function of expr*

Math Functions

abs (*expr*), *179*
annuity (*rate, periods*), *409*
atan (*expr*), *418*
average (*list*), *180*
charToNum (*expr*), *352*
compound (*rate, periods*), *407*
cos (*expr*), *412*
exp (*expr*), *400*
exp1 (*expr*), *400*
exp2 (*expr*), *400*
ln (*expr*), *422*
ln1 (*expr*), *422*
log2 (*expr*), *423*
max (*list*), *403*
min (*list*), *403*
random (*expr*), *178*
round (*expr*), *181*
sin (*expr*), *414*
sqrt (*expr*), *405*
tan (*expr*), *416*
trunc (*expr*), *182*

String Functions

length(*expr*), *359*
numToChar(*expr*), *352*
offset(*string1, string2*), *367*
result(), *455*
value (*expr*), *155*

Mouse Functions

clickloc(), *236*
mouse(), *239*
mouseclick(), *241*
mouseH(), *322*
mouseV(), *233*
mouseLoc(), *232*

Keyboard Functions

optionKey (*the*), *250*
commandKey(), *250*
shiftKey(), *250*

Time Functions

seconds(), *392*
secs(), *392*
ticks(), *392*
time(), *385*

Number of Functions

number of *chunks* in *factor*, *136*
number of *objects*, *136*
the number of cards of *bkgndDescriptor* (1.2), *136*

Found Text Functions (1.2)

the foundText
the foundChunk
the foundLine
the foundField

Selected Text Functions (1.2)

the selectedText, *154*
the selectedChunk, *154*
the selectedLine, *154*
the selectedField, *154*

Miscellaneous Functions

diskspace(), *648*
version(), *456*
the [long] version [of HyperCard] (1.2), *457*
the version of *stackDescriptor* (1.2), *457*
screenRect() (1.2), *583*
tool(), *501*
sound(), *483*
the [modifer] date, *382*

Error Function

result(), *455*

Property Functions

target(), *454*
tool(), *500*

Parameter Functions

params(), *447*
param (expr), *448*
paramcount(), *449*

4. Constants

down, *239*
empty, *349*
false, *241*
formfeed, *349*
linefeed, *349*
pi, *167*
quote, *349*
return, *349*
space
tab, *349*
true, *241*
up, *239*
zero, one, two, three, four, five, six, seven, eight, nine, ten, *167*

5. System Messages

arrowKey, *262*
closeBackground, *202*
closeCard, *202*
closeField, *202*
closeStack, *202*
controlKey, *263*
deleteBackground, *202*
deleteButton, *202*
deleteCard, *202*
deleteField, *202*
deleteStack, *202*
doMenu, *220*
enterInField (1.2), *210*
enterKey, *211*
functionKey, *263*
idle, *217*

(continued on last text page)

The Waite Group's

HyperTalk™ Bible

HOWARD W. SAMS & COMPANY
HAYDEN BOOKS

Related Titles

The Waite Group's Advanced C Primer++
Stephen Prata

The Waite Group's C++ Programming (Version 2.0)
Edited by The Waite Group

The Waite Group's Microsoft® C Programming for the PC, Revised Edition
Robert Lafore

The Waite Group's Turbo C® Programming for the PC, Revised Edition
Robert Lafore

The Waite Group's MS-DOS® Bible, Second Edition
Steven Simrin

The Waite Group's MS-DOS® Developer's Guide, Revised Edition
John Angermeyer, Kevin Jaeger, et al

The Waite Group's Understanding MS-DOS®
Kate O'Day and John Angermeyer

The Waite Group's Tricks of the MS-DOS® Masters
John Angermeyer, Rich Fahringer, Kevin Jaeger, and Dan Shafer

The Waite Group's Discovering MS-DOS®
Kate O'Day

The Waite Group's MS-DOS® Papers
The Waite Group

The Waite Group's OS/2™ Programmer's Reference
Asael Dror

The Waite Group's UNIX® Communications
Bart Anderson, Bryan Costales, Harry Henderson

The Waite Group's UNIX® Papers
The Waite Group

The Waite Group's UNIX® System V Bible
Stephen Prata, Donald Martin

The Waite Group's Advanced UNIX®— A Programmer's Guide
Stephen Prata

The Waite Group's Tricks of the HyperTalk™ Masters
(forthcoming)
The Waite Group

For the retailer nearest you, or to order directly from the publisher, call 800-428-SAMS. In Indiana, Alaska, and Hawaii call 317-298-5699.

The Waite Group's HyperTalk™ Bible

Mitchell Waite
Stephen Prata
Ted Jones

HAYDEN BOOKS
A Division of Howard W. Sams & Company
4300 West 62nd Street
Indianapolis, Indiana 46268 USA

©1989 by The Waite Group, Inc.

FIRST EDITION
FIRST PRINTING—1989

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-672-48430-7
Library of Congress Catalog Card Number: 89-63209

From The Waite Group, Inc.
Development Editor: *Mitchell Waite*
Content Editors: *Mitchell Waite and Kay Nelson*
Managing Editor: *Scott Calamar*
Technical Reviewer: *Rod Riggenback, Apple Computer*

From Howard W. Sams & Company
Acquisitions Editing: *Greg Michael*
Development Editing: *James Rounds*
Illustration: *Wm. D. Basham and David Cripe*
Cover Art Direction: *Celeste Design*
Cover Illustration: *Ron Troxell*
Indexing: *Ted Laux*
Keyboarding: *Lee Hubbard, Type Connection, Indianapolis*
Composition: *Shepard Poorman Communications Corp.*

Printed in the United States of America

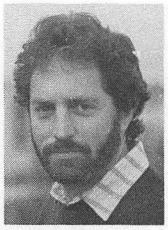
Trademark Acknowledgments

All terms mentioned in this book that are known to be trademarks or service marks are listed below. In addition, terms suspected of being trademarks or service marks have been appropriately capitalized. Howard W. Sams & Company cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

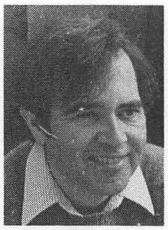
Apple, the Apple logo, HyperCard, ImageWriter, and LaserWriter are registered trademarks of Apple Computer, Inc.
Compuserve is a registered trademark of CompuServe Information Services, an H & R Block Company.
Excel, Microsoft Word, and Microsoft Windows are trademarks of the Microsoft Corporation.
4th Dimension is a trademark of Acius.
FullWrite is a trademark of Ashton-Tate.
IBM PS/2, OS/2, and Presentation Manager are registered trademarks of International Business Machines Corporation.
Icon Factory is a trademark of Hyperpress Publishing Company.
Macintosh, Macintosh Plus, Macintosh SE, Macintosh II, HyperTalk, Finder, MultiFinder, and Stackware are trademarks of Apple Computer, Inc.
MacPaint, MacDraw, and MacWrite are registered trademarks of CLARIS Corporation.
MacRecorder is a trademark of Farallon Computing, Inc.
MockWrite is a trademark of CE Software.
Radius Two Page Display is a trademark of Radius, Inc.
WordPerfect is a registered trademark of WordPerfect Corporation.

To Wavy Gravy, Ken Kesey, Bob Dylan,
John Lennon, Steve Jobs, Howdy Doody,
and all the other oddballs and great thinkers
that have crossed my path.

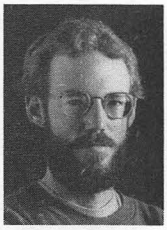
Mitchell Waite



Mitchell Waite is president of The Waite Group, a developer of computer books. He is an experienced programmer fluent in a variety of computer languages, including C, Pascal, BASIC, Assembly, and HyperTalk. He wrote his first computer book in 1976, and is co-author of *C Primer Plus*, *Microsoft QuickC Programming*, *UNIX Primer Plus*, and many other titles. Mr. Waite also writes a column on HyperTalk for *Hands On Macintosh*.



Stephen Prata is a professor of physics and astronomy at The College of Marin in Kentfield, California, where he teaches UNIX and the C language. He received his B.S. from the California Institute of Technology and his Ph.D. from the University of California, Berkeley. His association with computers began with the computer modeling of star clusters. Dr. Prata is co-author of *C Primer Plus*, *UNIX Primer Plus*, and *Microsoft QuickC Programming*, and author of *Advanced C Primer++*.



Ted Jones is an experienced programmer in Microsoft-BASIC, who recently switched to HyperTalk. He has a B.S. in liberal arts from Arizona State University. Mr. Jones chose to learn HyperTalk to avoid learning Pascal. He contributed to *The Waite Group's Tricks of the HyperTalk Masters*, and is a HyperTalk consultant and occasional contributor to the BMUG newsletter.

Overview

Introduction, 1
1 Starting, 29
2 Objects, 57
3 Audio and Visual Events, 101
4 Containers and Values, 127
5 Simple Math, 161
6 Messages, 197
7 Mouse, 229
8 Keyboard, 247
9 Ifs, 269
10 Repeats, 307
11 Strings, 341
12 Date and Time, 379
13 Math Functions, 399
14 Advanced Commands and Functions, 427
15 Advanced Sound, 461
16 Simulation, 487
17 Graphics, 521
18 Global Properties, 547
19 File Input and Output, 591
20 XCMDs and XFCNs, 613
A HyperCard-HyperTalk Version History, 645

Contents

Acknowledgments xxiii

Foreword xxv

Preface xxvii

To the Reader xxix

Introduction 1

What Is HyperCard? 3
Enabling Technology 3
Not an Application 4
Warming up Cold Media 5
HyperMedia ToolKit 5

Where Did HyperCard and HyperTalk Come From? 5
Authoring System or Programming Language? 6
Not Wild about WildCard 7

Principles of the Macintosh 9
The Macintosh Architecture 9
The Interface 9

The Nature of HyperTalk 15
The Problem with Traditional Languages 15

HyperCard Tricks for Better Scripts 20
Keyboard Alternatives 21
Power Keys 22
Keyboard Manipulation 22
Windoid Controls 22
Fast Info 22

Text Selections and the Clipboard 23
New Shortcuts 23
Something New: Peeking 23

Resources 23
Official HyperCard Utilities 23
Networks 24
Magazines 25
Books 25

1 *Starting* 29

What Version of HyperCard? 31

User Level and Preferences 31

Test Stack: HyperTalk Bible 33

The* script *Property and the Script Editor 35
Your First HyperTalk Script 35
The *beep* Command 37

Case Rules 38

Using the Script Editor 38
Cutting, Copying, and Pasting 39
Searching and Finding 39
Automatic Formatting of Scripts 40
Option-Return for Long Lines 40

The HyperCard-HyperTalk Relationship 41

Where's the Script? 42

Getting the Message 43

Making Comments 43

The Message Box—Your Scripting Control Panel 44

The HyperTalk Vocabulary 45

***Vocabulary Expansion:* put** 46
The Syntax 46
Using *put* and the Message Box 47
More *put* and a Trick 49
Switching between Tools 50

***Vocabulary Expansion:* set** 50

Summary 52

Review Questions 52

Exercise 55

2 Objects 57

HyperCard Object Refresher 59
- Cards 59
- Buttons 62
- Fields 63
- Graphics 64
- HyperCard's Object Layers 65
- Background and Card Layers 65
- Using Elements of a Card 67
- The Stack 67
- The Rest of the Story 67

Object Hierarchy: No Equal Opportunity 70

Passing the Message 74

A Humble Galactic HyperCard Metaphor 78

Naming Objects 79
- Names 80
- Relative Numbers 81
- ID Numbers 82

Object Hide and Seek 82
- Pop-Up Note Fields 84
- Pop-Ups That Work Better 84

***Using* me, target, *and* this** 85
- Getting to Know *me* 85
- *the target* of My Affection 86
- Now Hear *this* 87

What's in a Name? 88

Pathnames: Zipcodes for Stacks 89

Navigation 90
- *go* for It 91
- When *push* Comes to *pop* 94

Special Descriptors (Ordinals) 95

Summary 96

Review Questions 96

Exercises 99

3 Audio and Visual Effects 101

Let's Play: Introducing the* play *command 103

The* play tempo *Modifier 105

Visual Effects 106
The Dissolve Effect 107
Modifiers to Visual Effects 108
Venetian Blinds and Checkerboards 110
Movie Special Effects 1: Scroll and Wipe 111
Movie Special Effects 2: Zoom, Scroll, and Barn Doors 114

The* doMenu *Command 116

The* choose *Command 118

The* answer *Command 120
The *wait* Command 121
The *answer* Command Revisited 122

The* ask *Command 123

Summary 125

Review Questions 125

Exercises 126

4 Containers and Values 127

What's a Value? 129

What Are Containers and Variables? 129

More About Variables 131

Sources of Values 132
Strings 132
Numbers 132
Constants 134

Functions and Properties 136
the number of Function 136
Properties 137

Containers 138
Object Containers: Fields 138
Variables as Containers 141
the selection as a Container 152
Selection Functions 154
The Message Box as a Container 155

Summary 157

Review Questions 158

Exercise 159

5 Simple Math 161

The Math Operators 163
Addition in HyperTalk 164
Subtraction in HyperTalk 165
Multiplication and Division in HyperTalk 165

Using Arithmetic Operators in Scripts 166

HyperTalk Constants 167

Arithmetic Commands: A Programmer's Shortcut 168
The *add* Command 170
The *subtract* Command 171
The *multiply* Command 171
The *divide* Command 171
When Not to Use the Math Commands 172

Error Messages in HyperMath Land 172
*INF*inity 172
NAN: Not a Number 172

Advanced Operators 173
Whole Number Division: *div* 173
The Remainder Operator: *mod* 174
The Exponent Operator: ^ 174

Project Script: Poorman's Pseudoscientific Calculator 175

Extra Credit: Incrementing Your Calculator 176

Functions: Think of Them as Transformers 177
random: The Ultimate Abstractor 177
The Autocratic *abs* Function 179

Function Arguments: More Things to Care About 179
average: Just Another Pretty Function 180
Advanced Scripting: Romancing the Calculator 180
Evening It out: *round* 181
Dropping the Whole Thing: *trunc* 182

Precedence 183

The Large and Small of HyperTalk Numbers 185
It's in SANE 185
Advanced Scripting: Big Numbers in Scripts 186

***Masking the Result:* numberFormat** 187

Project Script: Chaos Theory and Verhulst 191

Summary 194

Review Questions 194

Exercises 195

6 Messages 197

A Quick Review 199

Introducing Parameters 201

Tale-Bearing Messages 202
- Messages from Your Mouse 202
- Mouse Button Messages 205

Messages to Fields 207
- Messages from Your Keyboard 209
- *returnInField* and *enterInField* 210
- Messages from Bigger Keyboards 210
- Messages from Objects 212

Combining Messages 214
- *startup* and Friends 216
- The *suspend* and *resume* Messages 216
- The *idle* Message 217
- The *help* Message 218
- The *quit* Message 218

The* pass *Keyword 218

Command Messages 220
- Menu Command Messages 220
- HyperTalk Command Messages 221

The* send *Keyword 223

Custom Messages and Handler Libraries 225

Messages in the Message Box 226

Summary 226

Review Questions 227

Exercises 228

7 Mouse 229

Where's the Mouse? 231
- The *the mouseH* Function 232
- The *the mouseV* Function 233
- The *the mouseLoc* Function 234
- The Hot Spot 236
- The *the clickLoc* Function 236
- The *the clickH* and *the clickV* Functions 238

What's the Mouse Up To? 239
- The *the mouse* Function 239
- The *the mouseClick* Function 241

The Simulated Mouse Click 243

Summary 245

Review Questions 245

Exercises 246

8 *Keyboard* 247

Keyboard Functions 249

The User Interface Functions 251
The *ask* Command 251
The *ask password* Command 254
The *answer* Command 257

Keyboard Messages 259
The *returnKey, enterKey,* and *TabKey* Messages 260
The *arrowKey* Message 262
The *functionKey* Message 263
The *controlKey* Message 263

Keyboard Simulation: The* type *Command 264

Summary 265

Review Questions 266

Exercises 266

9 *Ifs* 269

if then *Statements in HyperTalk—An Overview* 271

true false *Expressions (Boolean Expressions)* 272
Relational Expressions 274
is within 279
Using Boolean Expressions 280
Logical Expressions 282

The* if *Statements 290
The *if* Statement 290
Long Lines and an Alternative Syntax 292
Choosing Multiple Commands: *end if* 292
The *if then else* Statement 294
Variations of the *if else* Form 296
The *if then else if then else* Sequence 298

***Nested* ifs** 299
Data Checking 301
The *exit ⟨handler⟩* Command 301

Summary Script 303

Review Questions 305

Exercises 306

10 Repeats 307

Built-In Looping 309

Fixed Iteration Loops 309
The *repeat for* Control Structure 310
The *repeat with* Control Structure 312
Counting Backward 314

Nested Loops 316

Practical Nested Loops 319

Conditional Loops 320
The *repeat until* Control Structure 320
The *repeat while* Control Structure 324
repeat until Versus *repeat while* 325
Messages and States 327

Modifying Loop Behavior 327
The *exit repeat* Command 328
The *next repeat* Command 329
Exiting the Whole Script 331
The *repeat forever* Command 333

The* wait *Command 334
The *wait until* Command 334
The *wait while* Command 335

Summary Script 336

Review Questions 338

Exercises 339

11 Strings 341

String Basics 343
An Unquoted String Can Be a Troublesome Thing 345
String Characters 346
Numbers and Strings 346

Joining Strings: The & and && Operators 346

HyperTalk Text Constants 348

ASCII Code and the ASCII Functions 352

Text Units: HyperTalk Chunks 355
Basic Chunks 355
Composite Chunks 357
Ranges 357
Chunks as Destinations 358

Counting Functions 359
The *length( )* Function 359
The *number of* Function 360
Word Counting with a Button 362

Strings Within Strings 363
Testing for Content: *contains* and *is in* 364
The *offset( )* Function 366

The Selection 369
Selection Functions 370
Selection Information 370
The *select* Command 372

Summary Script 373

Summary 376

Review Questions 376

Exercises 377

12 *Date and Time* 379

the date *Function* 381
Using *the date* Function 382
European Date and Time 384

Getting the* time *of Day from Your Macintosh 385

Combining Date and Time Information 387

Date Conversions 388

Measuring Elapsed Time:* the seconds *and* the ticks *Functions 392

Elapsed Time Conversions 395

Summary Script 396

Review Questions 397

Exercises 397

13 *Math Functions* 399

General Math Functions 401

Review: The *abs( ), average( ), round( ),* and *trunc( )* Functions 401
The *max( )* and *min( )* Functions 403
Review: The *random( )* Function 405
The *sqrt( )* Function 405

Financial Functions 407
The *compound( )* Function 407
The *annuity( )* Function 409
Mortgages 411

Trigonometric Functions 412
The Look and Feel of the Sine Function 414
Circular Motion 416
The *atan( )* Function 418

Exponential and Logarithmic Functions 419
The *exp( )* Function 419
The *exp2( )* Function 420
The *exp1( )* Function 421

The Logarithmic Functions 422

Function Behavior 423

Summary 424

Review Questions 424

Exercises 425

14 Advanced Commands and Functions 427

User-Defined Functions 430
Function Basics 430
Designing a Function 431
Multiple Parameters 434
Functions with No Parameters 436
Functions, Strings, and Numbers 436
A Closer Look at Return Values 438
Function Parameters Revisited 440

User-Defined Commands 441
Making and Using a Simple Command 443
Commands with Parameters 444
Multiple Parameters 445
Alternative Method for Processing Command Parameters: *the params* and *the param of* 446

The Hierarchy 450
Modifying Existing Commands 452
Modifying Function Behavior 453
Where Has Your Message Been? (Using *the target*) 454

the result *Function* 455
Finding *the version* of HyperCard 456
Enhancing the Version Function 457

Summary 458

Review Questions 459

Exercises 460

15 *Advanced Sound* 461

A Quick Introduction to Sound 463

The* play *Command Syntax 466
The *voice* and *tempo* Parameters 466
HyperTalk's Music Language 467
The *tempo* Parameter 473
The Sound of *silence* 474

Project Script: The Sound Lab 475
Building the Sound Lab 476
Using the Sound Lab 479

Special Sound Effects 480
Sweep Special Effects 481
Boing Special Effects 482
High-Tempo Special Effects 482
The *play stop* Command 482
Sound Synchronization and *the sound* 483

Summary 485

Review Questions 485

16 *Simulation* 487

The* doMenu *Command 489
Simulating Stack Menu Items 491
Simulating Background Menu Items 493
Simulating Card Menu Items 493
Simulating Field and Button Menu Items 494
Simulating Navigation Menu Items 498
Simulating Painting Menu Items 498
Simulating Miscellaneous Menu Items 498

Simulation with the* type *Command 499

Simulation with the* choose *Command 500

Simulation with* the tool *Function 500

Simulation with the* set *Command and Object Properties 502

Stack and Background Properties 504
Button and Field Properties 504

***Using the New Rectangle Properties and* is within** 509

Simulation with the* click *Command 511

Simulation with the* drag *Command 512

The* dragSpeed *Property 514

Summary Scripts 514

Review Questions 517

Exercises 518

17 Graphics 521

Simulating with the* choose *Command 524
Simulating the Applicator Tools 524
Simulating the Shape Tools 534
Simulating the Manipulator Tools 541

Simulation Odds and Ends 549
Menu Commands to Show Dialog Boxes 549
Showing and Hiding Pictures 551

Summary Script 554

Review Questions 556

Exercises 556

18 Global Properties 547

Global Properties 559
User-Preference Properties 560
The *cursor* Property (Version 1.1 and earlier) 565
The *cursor* Property 566
The *dragSpeed* Property 567
The *editBkgnd* Property 569
The *language* Property 570
The *lockScreen* Property 570
The *lock screen* and *unlock screen* Commands 572
The *lockRecent* Property 572
The *lockMessages* Property 573
The *userModify* Property 575

Window Properties 577
The *location* Property 578
The *rectangle* Property 581
the screenRect Function 582

The *visible* Property 583
More Rectangle Properties 584

Summary Script 586

Review Questions 588

Exercises 589

19 File Input and Output 591

Working with Files or . . . I Don't Think We're in Kansas Anymore 593

Opening Files 594

Writing Data to Files 595

Delimiters 596

Reading Data to Files 596

Closing Files 598

The Plot Thickens 599

Simple File I/O Examples 599
Writing to a Sample File 600
Generalizing File I/O 600
A More Rigorous Example 601
Storing Chunks with Multiple Fields 603

Real World File I/O Examples 604

Summary 609

Review Questions 610

Exercise 611

20 XCMDs and XFCNs 613

How Does an XCMD/XFCN Work? 615
The Heap 616
How Data Is Stored 616

The Best of Both Worlds 618

Callbacks 618

What You Need to Get Started 622
Inside Macintosh 622
Macintosh Technical Notes 623
HyperCard Developers' Toolkit 623
MacTutor 623

How to Build Your Own XCMDs-XFCNs 624

What You Can and Can't Do 624
Hypercard's Heap 626
File I/O 627
Using *Get/WaitNextEvent* 627
XCMD/XFCN's Variables 627

The XCmdBlock 628

The* ChangeCase *XFCN Example 630

ChangeCase *Source Code* 632
Variables and Other Declarations 632
ChangeCase (Main) 634
DoSimpleCase Procedure 636
DoComplexCase Procedures 636

Installation 637

What Can You Do? 637

ChangeCase.p *Source Code* 639

A Version History 645

Features Added to HyperCard Version 1.2 and Related to Hypertalk 645

Features Added to HyperCard Version 2.2.2 and Related to Hypertalk 646
Enhancements to HyperCard 1.2.2 Related to Scripts 646
Bug Fixes Related to Scripts 647

Answers to Review Questions 649

Answers to Exercises 659

Index 675

Acknowledgments

I would like to take this opportunity to thank the people who have helped to make *HyperTalk Bible* a reality. First, I would like to thank Ted Jones and Stephen Prata for their alacrity, tenacity, faultless writing, and ability to endure my demands for perfection. When I met Ted Jones, I knew little about HyperTalk. I am in debt to him for much I have learned about this exciting language, and I hope I have taught him something about writing. Stephen Prata is a long-time and admired Waite Group author. His ability to so rapidly absorb the HyperTalk language, inspire others to understand it, and do all this under a deadline deserves praise. I would like to thank Apple Computer's Steve Maller for his chapter on writing external commands and functions (Chapter 20), for the *ChangeCase XCMD*, and for the nice evening at Gorilla Sushi. Steve is one of the most creative Macintosh programmers I have met, and I predict he will go on to create great products.

Kay Nelson, editor and author of several books, did a great deal of editing of the first and second drafts of *The Waite Group's HyperTalk Bible*, especially my early chapters, and I would like to thank Kay for being a most superb editor. Her ability to see the overall structure of a chapter and communicate that to the author is extraordinary.

I would like to thank the numerous reviewers that pored over *The Waite Group's HyperTalk Bible's* 1200 manuscript pages and gave us valuable feedback. Rod Riggenback, instructor at Apple Computer, helped us tune up the first part of the book. Peggy Redpath, evangelist at Apple, located the right people to look at the book, and David Szetela, Manager of Developer Services, opened the door and got our reviews started. Gary Wells, Interface Specialist for AutoDesk, tested every page of the book and learned HyperTalk from it. I am in debt to him for his care and passion in this project. Gary had a lot to do with my realization of the importance of design in a stack and his impact can be seen in the stack for our upcoming book *Tricks of the HyperTalk Masters.*

Scott Calamar was very helpful in pulling the final manuscript together and coordinating the three authors and all our meetings. Finally, I give my thanks to the people behind the scenes at Howard W. Sams, who took our manuscript and turned it into a marketable product that we are all proud of: to Jim Hill for his faith in the idea of a HyperTalk Bible, to Greg Michael for his constant pressure to do a book in

the Hayden Macintosh Library, to Wendy Ford for skillfully managing a most complex production job, to Ron Troxell for the book's great cover painting and the smirk on the handsome face of the wizard, to Jim Rounds for casting off this backbreaker of a manuscript, to Glenn Santner for bringing the vision of the cover to the artists, to Jim Irizarry and his sales team for moving our titles into the book stores in ever increasing quantities, to Damon Davis and Tom Surber for steering Howard W. Sams so wisely over the years, and to all the other people at Howard W. Sams who in one way or another were involved with making *The Waite Group's HyperTalk Bible* a success.

Mitchell Waite

I'd like to thank Mitch and Ted for introducing me to HyperCard.

Stephen Prata

I would like to thank Mitch Waite, first of all, for giving me the opportunity to contribute to this book. I would also like to thank Raines Cohen, Rob Smith, Scott Kronick, Jeremy Ahouse (and everyone else at the BMUG Scripting SIG), Jeff Stodard, and Mike Westphal for being ever helpful fellow travelers on the HyperTalk road. My contribution to the HyperTalk Bible would have been impossible without the combined efforts of George Boole, Clive Sinclair, Bill Atkinson, Frank Wiley, and my parents. Finally I would like to acknowledge my debt to the example of Richard Feynman who demonstrated that humor and common sense have a place even in technical subjects like mathematics.

Ted Jones

A Memo from:
Lewis Research Corp

108 E. Fremont Avenue Sunnyvale, CA 94087 (408) 730-5829

To: Reader

From: Jan Lewis

Re: *The Waite Group's HyperTalk Bible*

When Mitch Waite asked me to review his new *HyperTalk Bible*, my first reaction was "not another HyperCard book . . . there are already too many." I had watched publishers scramble to fill the void created by the projected 2 million owners of this product, and my shelves were already crammed with too many titles. The big problem with all these books: they lacked depth. The price for writing any book too fast is it's not likely to answer the real questions that arise in your everyday problem solving.

Now Apple has managed to give the impression that anyone can easily build complicated stacks that look like, act like, and are real applications and that anybody who had something to say, some expertise, some imagination, could turn out well-designed and esthetically pleasing stacks. And nearly anybody can, but it turns out that this task is not trivial. It takes learning a real computer language—HyperTalk. HyperTalk is an easy, elegant language that lends itself to writing small, legible chunks of code. But it is still a computer language.

So here we have this revolutionary product that is supposed to make building applications and personalizing information spaces easy, and these books that are not telling us how to accomplish that goal. People are getting stuck on the HyperTalk end of things. Mitch tells me that this book was especially designed to teach me HyperTalk in a lesson format, using step-by-step examples. Suspecting that he had tried to outdo all the book products on my shelves, I sat down to see if this book could make a difference in our results with HyperCard.

I'm back, and I'm happy to say the answer is yes. *HyperTalk Bible* does what it sets out to do—provides us with a simple and easy-to-follow course in HyperTalk. The book is beautifully laid out, making HyperTalk easy to learn, even for the first-time programmer. It does assume you have played with HyperCard and understand it at the authoring level. The book first teaches new concepts, such as objects, hierarchies, and message handling, so you get a firm foothold in these new aspects of HyperTalk programming. The book takes a structured, example-driven, "do it and learn" approach. You'll find that every chapter is loaded with numbered examples to follow. There are neat projects in the book that show off more complete uses of HyperCard. *SoundLab*, for example, is a great way to learn the ins and outs of the *play* statement, and I think you will get some good ideas for building sounds into stacks.

By now, you can probably tell that I like this book. HyperCard and HyperTalk offer something no other system of tools and software does: this wonderful way for stacks to be "crafted" so as to use fewer and fewer lines of code. There is a sort of graceful way of writing HyperTalk and I think this book has captured that subtle aspect of the language, while at the same time teaching us how to use it. I'd say buy this book.

Jan Lewis is the founding editor of *HyperAge*, a Stack Hacker Supreme, and a confessed "HyperCard Addict." More than a million people regularly read her syndicated column, "But Jan Lewis Says," and see her regular appearances on the weekly television show "Computer Chronicles." Ms. Lewis's firm, Lewis Research Corp of Sunnyvale, California, provides HyperCard design and consulting services to a variety of developers and end-users.

Preface

During the last twelve years, I have written teaching computer books on BASIC, Pascal, C, Logo, and 680x0 and 80x86 assembly language. I can honestly say that after brewing up hundreds of programs in all these languages, nothing has come close to motivating me as much as HyperTalk. It is an incredible tool and a lot of fun. It is so inspiring that Ted Jones, an avid Macintosh BASIC programmer and coauthor of this book, jumped the BASIC ship immediately when HyperCard was born. Stephen Prata, a C programming master who teaches UNIX and C at our local college and authored our best-selling *C Primer Plus* is also taken with HyperTalk.

HyperTalk is the language of the popular HyperCard product now supplied with every Apple Macintosh computer. It is the first programming language for many Macintosh users, so we designed *HyperTalk Bible* to be all you need to become a proficient HyperTalk programmer. We wrote *HyperTalk Bible* with these goals in mind:

- Make it a self-study book that teaches programming concepts along with HyperTalk syntax.
- Make the material accessible to beginners so that everyone can learn HyperTalk, regardless of their background in programming.
- Stress modern programming concepts of objects and messages early in the book.
- Cover the entire scope of HyperTalk, including every subject and every instruction, even external commands and functions.
- Supply short, useful examples that give the reader hands-on solutions to real-life problems.
- Give the reader an easy learning path, with later chapters building on previous chapters, reenforcing early learning, and providing a sensible context for each new lesson.
- Provide quizzes and exercises to make the book useful for a first semester course on programming with HyperTalk.
- Provide project scripts that point the direction to building more elaborate HyperCard stacks.

- Provide a companion disk through Heizer Stackware, containing all scripts, resources, icons, sounds, fonts, and graphics from the book, so the reader can easily elaborate on our stack projects and examples (see order card for ordering instructions).

The authors spent six solid months each writing this book. In fact, in the process, The Waite Group also developed a more advanced book called *The Waite Group's Tricks of the HyperTalk Masters* that you might want to read after this one. As leader of the writing team, my principal experience with HyperTalk came from designing a commercial stack that serves as an interface to a set of HyperTalk tricks.

If you want to send us comments on the book or suggestions for improvements, or if you wish to contribute to our future Macintosh books, you can reach us at: The Waite Group; 100 Shoreline Highway, Building A, Suite 285; Mill Valley, California 94965; (415) 331-0575. If you have a computer and modem, you can also reach me on the following networks. (See the Introduction for more details.)

Well: hplabs!well!mitch
AppleLink: D2097
Compuserve: 75146,3515
BIX: mwaite
MCI Mail: MITCH

We hope you enjoy the *HyperTalk Bible*.

Mitchell Waite

To the Reader

This section provides information about this book, what it can do for you and how you can best use it. It also describes how to order the HyperTalk Bible stack and explains the conventions used throughout this book.

Is This Book for You?

HyperTalk Bible is designed for the person who wants to create powerful and impressive HyperCard stacks. Stacks are a new area of Macintosh computer application software developed by Bill Atkinson of Apple Computer. Stacks run under HyperCard somewhat as a cassette tape runs on a tape recorder and can be created and manipulated by using various "tools" that come with the product. Stacks are built around several important "software objects" that mimic objects found in the real world. For example, you can quickly create the software equivalents of real buttons and switches, movable text fields, and cardlike objects. You can link these cards and buttons. You can place beautiful artwork on the cards with graphics tools. Manipulating HyperCard this way is called *authoring*. To really make your stacks powerful, however, you need to learn a special language called HyperTalk that Apple developed and built into HyperCard. HyperTalk lets you "program" the objects that HyperCard depends on and therefore allows them to perform in powerful ways.

Are You a HyperCard Author?

We designed the *HyperTalk Bible* to make learning HyperTalk easy and fun. You should understand HyperCard at the authoring level (know how to create simple buttons, make card graphics, make links, and so on). If you have never programmed a computer before, it's okay. Occasionally, we make references to other languages in case you are familiar with BASIC, Pascal, or C. If you are familiar with programming languages in general, you may be able to skip some of the early sections of the book. You might want to skim these sections, however, to find some of the unique features of HyperTalk.

Object-Oriented Programming

HyperTalk stands out from previous computer languages in that it uses what are referred to as *object-oriented programming* (OOP) techniques. OOP is a new area of computer-science based on manipulating entities that have properties and characteristics, and it requires a different kind of thinking about what constitutes a "program." These new ideas and tools call for a new paradigm to the teaching of programming, and in writing this book, the authors reveal some of the general principles of good object programming that make up this paradigm. For example, as you learn HyperTalk, you also learn about the goal of generalization of your scripts so that they can be reused in any object and in any stack.

Learn by Doing

To make learning HyperTalk enjoyable, *HyperTalk Bible* is completely hands-on and each concept has an example you can type and try if you wish, or you can just follow along mentally. We also include numerous stack project scripts that combine the ideas of previous concepts and commands.

Although the chapters are designed to be read serially, and they build on previous knowledge, we wrote this book so that you can open to any chapter and find all you need to know in one place without having to skip around in the book. The price of this approach is redundancy in some places and we may explain a command or concept more than once.

Illustrations and Exercises

This is a visual book. Many scripting concepts are best explained with pictures, so we often use graphics to support the ideas in the text. Screen dumps are used so that you can see exactly how things will look. In the first few chapters, we tell you every tool to choose to make your examples work. Each chapter contains a complete set of quizzes and exercises. Use these to make sure you understand the ideas before moving to the next chapter. The answers to all review questions and exercises are listed at the end of the book.

The Bible Stack

All the scripts in this book are available in a complete stack called *The Waite Group's HyperTalk Bible* stack. It is available from Heizer software at the following address:

The Stack Exchange
Heizer Software
P.O. Box 232019
Pleasant Hill, CA 94523
800/888-7667

Ordering instructions are provided on the last page of this book and there is a tear-out order card also included.

The stack contains all the screen images, icons, bit-maps, fonts, external commands, and sounds described in the book. Having the stack not only gives you these extras, but makes learning HyperTalk much faster, because you don't bog down trying to be a typist or graphics designer. The stack will be periodically updated, so some features in it may not be in the book.

Conventions Used in This Book

Everything preceded by the words "Enter," "Type," or "Name it" is printed in a special font. For example: Name it `myButton`. The following are also printed in the special font:

`commands`	`properties`
`messages`	`statements`
`functions`	`modifiers`
`containers`	`constants`
`ordinals`	`descriptors`
`variables`	`text strings`
`parameters`	`operators keywords`

All hands-on practice sessions are called Examples and each is numbered sequentially within its chapter, so the fifth example in chapter 5 is called Example 5.

The following list illustrates the icons used in *HyperTalk Bible* and what they represent:

Version 1.2 Specific
This icon designates new features of Version 1.2

Hyper Warning
This is a mild note of warning, usually designating general HyperCard rules that you should keep in mind.

Master Warning
This icon describes really dangerous things to watch out for.

Hyper Facts
These are general notes, facts, or tips about HyperTalk.

Hyper Masters
This symbol represents an advanced HyperTalk or programming topic for hackers, BASIC masters, Pascal masters, or XCMD users.

Hyper Limits
HyperTalk does have its limits, as this icon illustrates.

Math Masters
These are math related tips and tricks for master thinkers.

Bookmark

HOWARD W. SAMS & COMPANY

DEAR VALUED CUSTOMER:

Howard W. Sams & Company is dedicated to bringing you timely and authoritative books for your personal and professional library. Our goal is to provide you with excellent technical books written by the most qualified authors. You can assist us in this endeavor by checking the box next to your particular areas of interest.

We appreciate your comments and will use the information to provide you with a more comprehensive selection of titles.

Thank you,

Vice President, Book Publishing
Howard W. Sams & Company

COMPUTER TITLES:

Hardware
- ☐ Apple I40
- ☐ Macintosh I01
- ☐ Commodore I10
- ☐ IBM & Compatibles I14

Business Applications
- ☐ Word Processing J01
- ☐ Data Base J04
- ☐ Spreadsheets J02

Operating Systems
- ☐ MS-DOS K05
- ☐ OS/2 K10
- ☐ CP/M K01
- ☐ UNIX K03

Programming Languages
- ☐ C L03
- ☐ Pascal L05
- ☐ Prolog L12
- ☐ Assembly L01
- ☐ BASIC L02
- ☐ HyperTalk L14

Troubleshooting & Repair
- ☐ Computers S05
- ☐ Peripherals S10

Other
- ☐ Communications/Networking M03
- ☐ AI/Expert Systems T18

ELECTRONICS TITLES:
- ☐ Amateur Radio T01
- ☐ Audio T03
- ☐ Basic Electronics T20
- ☐ Basic Electricity T21
- ☐ Electronics Design T12
- ☐ Electronics Projects T04
- ☐ Satellites T09
- ☐ Instrumentation T05
- ☐ Digital Electronics T11

Troubleshooting & Repair
- ☐ Audio S11
- ☐ Television S04
- ☐ VCR S01
- ☐ Compact Disc S02
- ☐ Automotive S06
- ☐ Microwave Oven S03

Other interests or comments: _____

_____

_____

Name _____
Title _____
Company _____
Address _____
City _____
State/Zip _____
Daytime Telephone No. _____

A Division of Macmillan, Inc.
4300 West 62nd Street
Indianapolis, Indiana 46268

48430

Bookmark

HOWARD W. SAMS
& COMPANY

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 1076 INDIANAPOLIS, IND.

POSTAGE WILL BE PAID BY ADDRESSEE

HOWARD W. SAMS & CO.
ATTN: Public Relations Department
P.O. BOX 7092
Indianapolis, IN 46209-9921

Please send me *HyperTalk Bible: The Stack* (#30243) and a free copy of the PolyButtons Stack.

Name_____ *Office Use Only:* _____

Daytime Phone (_____) _____

Diskette Order Form

HyperTalk Bible: The Stack	$ 19.95
CA residents add $1.30 sales tax	$ _____
Shipping and Handling *US/Canada - $3 / Foreign - $7*	$ 3.00 *or* $7.00
PolyButtons Stack	$ **FREE**
Heizer Software Catalog	$ **FREE**
Total	$ _____

Payment:

Check enclosed (payable to Heizer Software) # _____

VISA MasterCard American Express

Card #_____ Exp. Date_____

Signature _____

PLEASE PRINT:

Name_____

Organization_____

Address_____

City_____ State_____ Zip_____

Place
Stamp
Here

Heizer Software

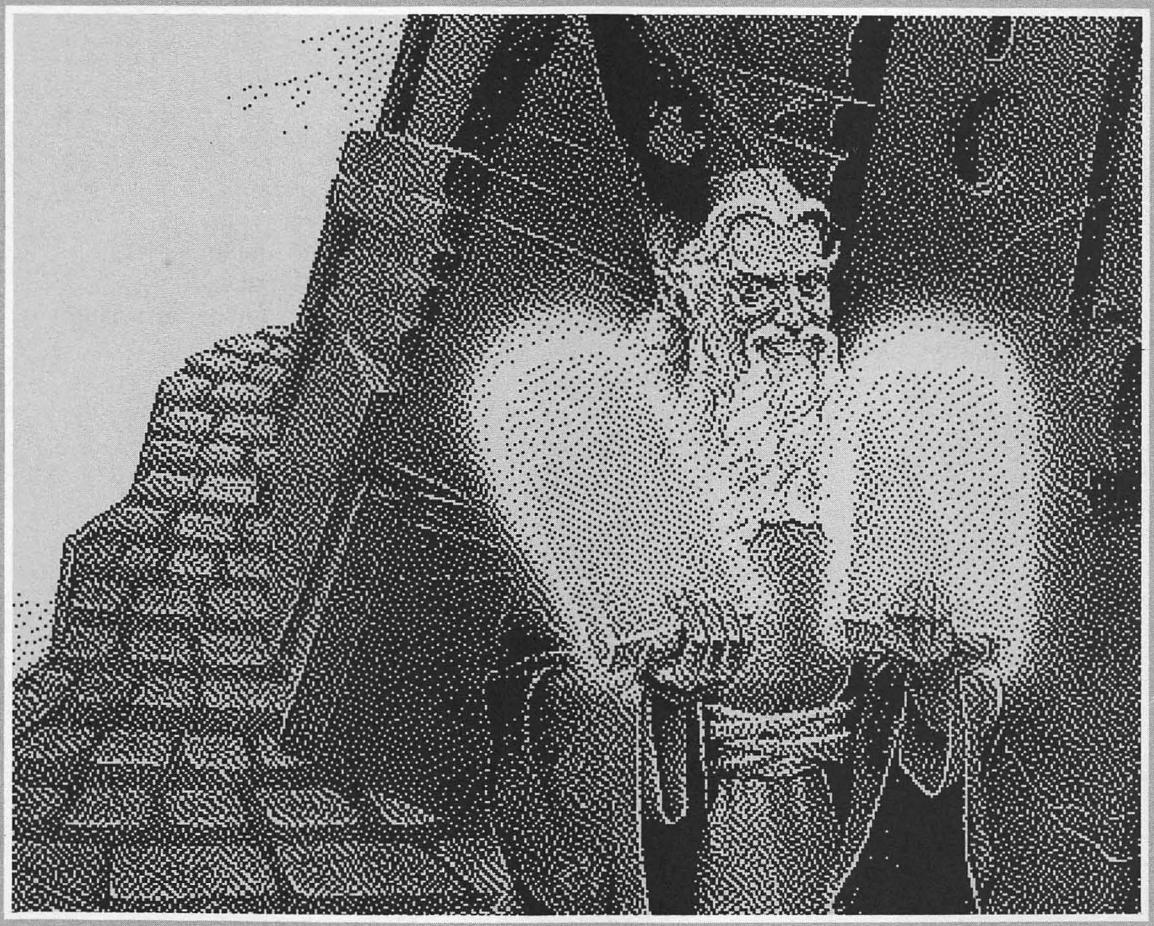
P.O. Box 232019
Pleasant Hill, CA 94523

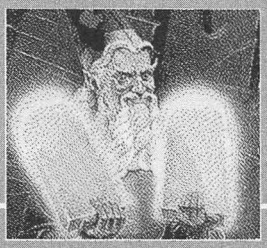
USA

HyperTalk Bible: The Stack

Please fold along line and tape closed. Do not staple.

Introduction





Quick View

Concepts

What is HyperCard?
Where did HyperCard and HyperTalk come from?
Authoring system
Software erector set

Concepts *cont'd*

HyperMedia
Principles of the Macintosh
The nature of HyperTalk
HyperCard tricks for better scripts

Introduction

Welcome to *The Waite Group's HyperTalk Bible*, a book designed to teach you how to use HyperTalk, the internal language of HyperCard.

What Is HyperCard?

If you already know HyperCard, you may want to skip this section. If you are new to HyperCard, do not understand its features, or have not reviewed its documentation, this section helps you understand HyperCard's purpose, capability, and future.

Enabling Technology

Someone looking at HyperCard for the first time sees a set of small graphic pictures sitting on a rectangular cardlike image on the computer's screen. Figure I-1 shows a HyperCard Home stack customized by the author.

Although images play a large role in what HyperCard is, it is much more than images on the screen. Simply put, HyperCard is a friendly, easy-to-use, software "erector set." HyperCard allows you to "roll your own" Macintosh software applications (called *stacks*) to look exactly as you wish. You build functionality into these stacks that, in some cases, rivals full-blown, professional applications. Consider HyperCard a construction kit with software parts that you control, move, program, and train to do your bidding. This quality of HyperCard that enables you to craft your own applications gives it a nature spanning several traditional areas of software. Because HyperCard is so general, you may hear it called a database, information manager, interface builder, help system, and financial manager. Indeed, a popular Macintosh magazine survey of its more than 300,000 Macintosh users found that HyperCard was voted best product in four areas.

You have probably heard that writing software for the Macintosh is a difficult undertaking, much harder than for text-based computers, and that you have to be a programmer "guru" to build complex custom systems for filing, calculating, or database control. Although it's true that there are more friendly "macro" languages in user-programmable commercial products (like Excel or 4th Dimension), you still

Figure I-1. HyperCard Home stack on the Macintosh.

need to be a dedicated "hacker" to make these languages work. HyperCard shatters the notion of gurus, jocks, and wizards and is a complete software development tool for the nonprogrammer. You will hear HyperCard called an "enabling" technology because you may do things with the computer that were impossible for the average person just a short time ago.

Not an Application

HyperCard is not an application like a word processor, spreadsheet, or database program. These programs are specialized to perform one, and only one, function and to perform that function extremely well and with great speed. Consequently, such applications are written and finely tuned in a very complex language (usually assembly language— the language of the microprocessor chip that runs the Mac). Because a word processor doesn't also have to work as a database program, it can be optimized for manipulating words. HyperCard, on the other hand, is designed to let you build a myriad of applications.

HyperCard Not for Building Applications
Although a HyperCard stack will not rival a focused application in performance or speed, and it would be foolish to build a word processor in HyperCard, a HyperCard stack can be made to "look and feel" very much like a word processor, a database, or whatever.

Warming up Cold Media

Perhaps what HyperCard does best is explained in the light of ideas put forth by Marshall McLuhan. HyperCard makes cold media hot. This means that HyperCard allows you to take large databases made up of lines of plain text characters and dress them up in a nice looking graphics interface, with "controls" to make accessing that text easier and more intuitive. What normally would be viewed as line after line of boring text becomes a beautiful easy-to-access graphic metaphor. For example, HyperCard can turn a real-estate-listing database into a simulation of visiting the actual house. Scanned images can be placed in the database, and HyperCard can present all the vitals on each house when the user clicks different objects on the screen. HyperCard gives a level of interactivity not found on traditional read-only, search-and-replace databases.

HyperMedia ToolKit

There is a reason for the Hyper in HyperCard. Hyper refers to HyperMedia, a concept of linking information by context. In a hypermedia "document" words and pictures can be linked to other documents. These linkages are enabled by an indicator device, such as a mouse, so that when you click a special word, your screen reveals new information about that word. The new information may "pop up" on top of the word, or your entire screen may be replaced with a new document and a way to get the screen back to where it was. Such linkages, if well designed, allow readers of the document to follow their own paths through the information, instead of proceeding in the linear order that most books—even this one—follow. In a HyperMedia universe, we would click a mouse button instead of turning pages.

Thus, HyperCard, besides being a software erector set, is a HyperMedia toolkit that you use to organize information in ways only dreamed of before. Indeed, one of the cornerstones of HyperCard is its linking mechanism.

Where Did HyperCard and HyperTalk Come From?

In 1984, HyperCard was a dream in the mind of Bill Atkinson, one of the early developers of the Macintosh computer. Atkinson was responsible for writing the first and most important application for the Macintosh: the famous MacPaint drawing program. This product helped propel the acceptance of the Macintosh, because it showed off many of the Mac's superior features over other computers, particularly in the area of graphics. Atkinson also was the developer of one of the most important internal features of the Macintosh: a special graphics language called *QuickDraw*. QuickDraw is embedded in the Mac ROM (read-only memory). Every application that runs on the Mac uses QuickDraw to write text or draw characters on the screen.

Given Bill Atkinson's place on the Mac development team, his experience in building MacPaint and watching people use it for several years, and his fertile and creative nature, he was in the perfect position to dream up a new kind of software for the Macintosh, one that would break all the traditional rules of what software was supposed to do.

Authoring System or Programming Language?

Early on, HyperCard was called WildCard, and there are, to this day, some reminders of this in the Art Ideas stack (look for the King card holding a Macintosh computer in one hand and a mouse in the other). See Figure I-2.

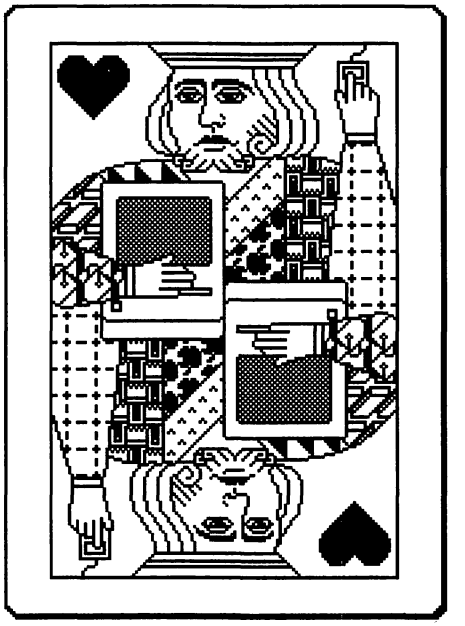


Figure I-2. WildCard from the Art Ideas stack.

In its early days, HyperCard was to be an *authoring* system that lets you create systems to teach and guide students and learners. In comparing HyperCard to other authoring systems, Bill Atkinson mentions Owl's Guide (more of a HyperMedia program), Plato, Pilot, and Logo. Owl's Guide is a HyperMedia tool for the IBM PC that allows you to create documents with linkages that you set up. Plato is an educational computer network with hundreds of learning programs and tools that teachers can use to craft these programs. It is not a commercial product, however. Pilot is known as an authoring language and is used for writing courseware, a kind of software

devoted to teaching, quizzes, and so on. Logo is a general computer language based on the concept of Turtle graphics. It has been popular in grammar and high schools for teaching concepts of geometry, programming, and graphics. HyperCard has attributes of all these programs, but it is much more.

To understand *why* HyperCard is more, consider that Bill Atkinson based HyperCard on some fundamental discoveries he made while watching people use computers and his programs over the years. First, he saw that people learn best when given information in small steps and that no one step should require a great amount of effort to understand. Thus, someone learning to use a programming system for the first time should be offered different layers of capability. Each layer should add more power to the user's set of tools, and the overhead of learning each new layer should be minimized. Atkinson's second observation was the merit in trading simplicity for complexity. If a method of doing something was too complicated to understand, he tossed it out. HyperCard represents an astonishing amount of distillation that you come to appreciate only after using it for some time.

Bill Atkinson describes four levels of user power that he wanted to embed in HyperCard. These start with *browsing*, moving around and viewing a database by clicking images on the screen. The next level is *painting*, using a MacPaint tool to draw graphics images. This takes more skill to master than browsing but yields equivalent new rewards. The third level is *cutting and pasting*, selecting areas of text or drawings and moving them to other locations. The fourth and final layer is a programming language for manipulation of objects that are created. When Atkinson started working on HyperCard, only the first three levels were considered. As his work progressed, it became clear to him that it was much more than he originally envisioned. With the addition of the HyperTalk language, HyperCard moved beyond mere authoring and became a complete development system for the Macintosh.

Not Wild about WildCard

HyperCard almost didn't make it out the door as an Apple product. Bill Atkinson felt that HyperCard had to be made available to millions of people and that if Apple just made it a new software product available through normal channels, much of the mass acceptance would be missed. (He had some poor experiences with his MacPaint program because Apple fumbled with how to be a hardware company and sell software at the same time. This problem has been addressed by Apple's spinoff, Claris.) Atkinson wanted Apple to "bundle" HyperCard with the Mac, meaning it would come with every Mac that was sold as part of the system software. There was much resistance at Apple and in the industry itself to this idea. Many third-party developers of Mac software felt Apple would be unfairly competing with them because their products were not bundled.

The story goes that Bill Atkinson put himself on the line with John Sculley, the chairman and CEO of Apple, requesting that Apple bundle the yet unfinished HyperCard. If Apple wouldn't bundle, Atkinson was willing to quit and find a company that

would push HyperCard in the way he wanted. Apparently, Sculley was convinced by Atkinson, and it appears it was a good decision. More than 1 million people are using the product, more than 200,000 stacks have been created, and a giant third-party support market is forming around the product.

Today, Atkinson claims that HyperCard is as important to the Macintosh as Apple BASIC was to the Apple II. Applesoft BASIC allowed thousands of enthusiasts to put computer technology to work. In essence, Apple gave each user a wrench and didn't tell them what to do with it. HyperCard is similar, except it offers a more complete workshop rather than just a single tool.

Other Credits

HyperCard was a team effort. People essential to HyperCard's development were Dan Winkler, who wrote the HyperTalk language that this book is about, Ted Kaehler who wrote the Sound Manager, Carol Kaehler who wrote the Help Stack, and Adam Paal who created the printing manager for HyperCard. Chris Espinosa, a long time Apple veteran, ran interference for Atkinson inside Apple's walls, evangelizing HyperCard to Apple's managers who did not see the vision of this product. Scot Kamins, former Waite Group author, wrote the *HyperCard User Guide*, and Kristee Kreitman, an artist well known for her icons and designs, created much of the artwork that you see in HyperCard.

Technical Achievements of HyperCard

HyperCard has made several technological achievements. The compression of bit-maps is critical with HyperCard because bit-maps form the basis of all graphics and take up much room in memory. Bill Atkinson achieved somewhere around a 30 to 1 packing ratio on HyperCard's bit-maps, so an 800K-byte stack could theoretically contain up to 2.4 megabytes of graphics. A CD ROM contains around 800 megabytes, so with the compression, you get over 2,400 megabytes.

Searching for text in HyperCard proceeds at a rate that is about 200 times faster than traditional methods employed today. Much of this work is proprietary to Apple, and patents may be in the application stages. Bill Atkinson has publicly stated that Apple will reveal the basic file formats for storing HyperCard stacks so that third-party developers can write stack-compatible applications. (These specs allow an MS-DOS or OS/2 version of HyperCard to read Mac stacks and vice versa.) In fact, a PC version of HyperCard, called LinkWay, has been announced by IBM. LinkWay has the same set of functions and features as does HyperCard, but its graphics, sounds, and tools are much more limited. It organizes data into pages and folders, instead of HyperCard's cards and stacks. Since Apple still hasn't made HyperCard's file formats public, LinkWay cannot read HyperCard's files.

Principles of the Macintosh

Most of you have used both the Macintosh and HyperCard before. However, because the HyperCard product is bundled with each Macintosh sold today, HyperCard represents the first interaction with the Macintosh computer for some of you. This can be unnerving because, although the Macintosh is an improvement over established computers in terms of its operation, many of the new ideas exemplified by its operating system can flabbergast even the most patient. This section is designed as a quick refresher course in the fundamental nature of the Macintosh system. Here, we present those concepts of the Macintosh that you need to know as you learn HyperTalk.

The Macintosh Architecture

The Macintosh deviates from computers of the past in several ways. All computers have some sort of custodian program called the *operating system* that does all the basic housekeeping tasks a computer needs, such as copying and listing files, erasing disks, and so on. Traditionally, this large program is provided on a disk and the computer won't start without loading it first. Most of the Macintosh operating system is built around what is called the Apple Toolbox, ROM in the computer that holds most of the system routines of the Macintosh operating system. The rest of the Macintosh operating system is located in the disk-based System file, which Apple updates periodically. Thus, every Macintosh computer comes with a tremendous amount of built-in intelligence.

The Apple Toolbox contains a large number of programs called *managers*. These are complete subsystems that perform all the complicated tasks that this machine requires. Application programmers have their programs access these managers, and that saves them from having to include all the fundamental code in their own programs. However, this code is still quite complex. For a HyperTalk scriptor, these managers will probably never be an issue.

The Interface

Perhaps the most overt difference between the Macintosh and traditional computers is the famous Macintosh *interface*. Indeed, this interface is so revolutionary that it has become the source of a legal battle between Apple and several large software companies. The interface refers to what the user sees on the screen when the computer is turned on, how the computer's resources are accessed, and, in general, the interactions between the person using the computer and its resources (keyboard, mouse, etc.). Macintosh has popularized several new concepts in computer interfaces. Let's review these.

Graphics

The most impressive feature of the Macintosh may be its graphics. Older computers could only display letters, numbers, and special symbols of the alphabet. Drawing

graphics and pictures required purchasing special hardware components to plug into the computer and switching your operating system to special graphic "modes" so that it knew these graphics existed.

Macintosh does not display text characters. Rather, everything you see on the screen is really made up of tiny squares that can be either black or white (or color on the Mac II). These are called *pixels*. The placement of pixels is used to "draw" letters and numbers, so the graphics mode is the only mode the Macintosh can be in. Graphics and pixels consume more memory than text, and that makes the Macintosh more dependent on large memory than older computers. The proliferation of graphics on the Macintosh is what makes Mac applications look so good compared with most older computers. Few graphics modes of other computers display pixels that are as crisp and easy to see as the Macintosh.

The use of graphics also makes the Macintosh more of a visually based computer than previous machines. Adding graphics in HyperCard is done through the painting mode. Indeed, the paint program supplied with HyperCard is actually superior to most stand-alone graphics applications. To use HyperCard to its fullest potential, you should become proficient at using the paint tools.

HyperCard on a MacPlus
Under MultiFinder, 750K is the absolute minimum memory configuration recommended for running HyperCard. On a 1M machine, running HyperCard limits you to the Typing user level. Processes that need more memory, like the paint tools, either beep and don't work or return you unexpectedly and rudely to the Finder.

Mouse and Cursor

The Macintosh computer uses the idea of "pointing" and "clicking" to manipulate programs. A device called a *mouse* comes with every Macintosh. The mouse is about the size of a bar of soap and attaches to the Macintosh by a cable. (Future Macs may, we hope, have a wireless mouse.) You "drag" the mouse on a flat surface and then point it at a specific location. A tiny ball in its base rotates like a wheel and this rotation is used to move a tiny arrow image on the screen. When you move the mouse, the arrow moves with it. This is called the *mouse cursor*. The mouse has a button that you can press (click). When the arrow is over certain areas, and you click the mouse button, certain results occur. Pointing and clicking is the cornerstone of much of the Mac technology.

HyperTalk allows you to program the cursor to change shape as it moves over different areas of the screen.

Windows and Cards

Every Macintosh application, including the operating system itself, uses *windows*, rectangular areas that display information, such as text or graphics. These windows can be moved on the screen, their size can be adjusted, and they can be organized to

overlap by pointing, clicking, and dragging with the mouse. (We leave the details of such operations to the *User Guide* supplied with the Macintosh.) Windows extend the metaphor of the Apple Macintosh *desktop*. This is an idea that the screen represents the top of a typical desk. You may have to stretch your imagination to make this metaphor work because many of the objects that make up the Macintosh interface have more in common with a dashboard than a typical desk.

Windows have parts that you maneuver with the mouse and control how the window's contents are displayed. For example, Macintosh windows have *scrolling bars* on their edges that let you move the information in the window up and down, or to the left and right. Windows can be closed by clicking a little box called the *close box*. A window also carries its name at the top of the window in the *title bar area* so that you can remember what each window refers to. Most applications allow several windows to be opened at a time.

HyperCard doesn't use windows. Instead it uses a fixed object called the *card*, a rectangular object that just fills a Macintosh Plus or SE screen (9 inches diagonally, about 4.5 inches high and 6.5 inches wide). These cards can't be scrolled or resized like regular windows. Bill Atkinson broke the tradition Apple had established over the years that all programs *must* use windows. Because the algorithms for moving bit-maps on a fixed-size card are much simpler and faster than those used on a variable size window, it is doubtful HyperCard will ever do windows. However, clever people have added external commands and functions that enable you to graft windows to cards. (Chapter 20, "XCMDs and XFCNs," covers external commands in more detail.)

Menus

The Macintosh gives users a way to select tasks from what is called the menu system. *Menus* are graphic elements that pop down when you click the mouse in an area called the *menu bar* found at the top of all Macintosh screens. Each menu bar contains a list of items that represent commands. Each application has its own set of menus. If the mouse is clicked and released over an item in a menu, the action described by that item is performed.

HyperCard has several menus. In fact, HyperCard adds new menus to the screen when different tools are selected. Unfortunately, you can't create custom menus for your stacks in HyperCard or HyperTalk without reverting to tricks.*

In newer computers that compete with the Macintosh interface, such as Sun's Open Look, every window has its own menu bar. This is actually superior to the Macintosh system because, with the Mac on a larger screen, the user must move the mouse a great distance to access the menu bar.

HyperCard features a new idea called *tear off menus*, menus that can be moved away from the menu bar and placed on the desktop where all the items on them are visible. HyperCard's Tools and Patterns menus can be torn off.

---

* See "Menu Fields" in *The Waite Group's Tricks of the HyperTalk Masters* for details on how to simulate different kinds of menus with pure HyperTalk (Indianapolis: Howard W. Sams & Company, 1989).

Icons

Icons are small rectangular images about 0.5-inch square. They usually contain graphics comprising 32 pixels vertically and horizontally. These images form much of the Mac interface. Icons are used to give a visual indication of a certain function. Often, the user can click them with the mouse to create certain actions. All Macintosh programs and files have a unique icon that identifies them and appears on the desktop.

HyperCard allows you to add icons to your stacks and cards easily. These icons are often protected by copyrights, so be careful when using them in commercial ventures. It is not clear whether the icons provided by Apple with HyperCard are in the public domain or the exclusive property of Apple computer. Strangely, HyperCard provides no tools for making custom icons—it just makes icons easy to move around. There are special programs you can buy for making your own icons, such as Icon Factory. Apple has a utility called ResEdit that lets you modify and create icons and is available on most computer networks.

Controls: Buttons and Scrolling Bars

Controls are objects on the Macintosh screen (buttons, check boxes, and scroll bars) that the user manipulates with the mouse. Controls can cause instant actions or they can change settings to modify a future action. *Button controls* cause an action when clicked or pressed. *Check boxes* retain and display a setting, either checked (on) or unchecked (off). Clicking with the mouse reverses the setting. Check boxes are frequently used to modify a future action. *Radio buttons* retain their on/off settings and are usually organized into groups with the condition that only one can be on at a time, like buttons on an older car radio. *Dial* is another kind of control. The only Macintosh example of a dial is the scroll bar.

HyperTalk allows you to create and manipulate buttons with ease but does not allow you to create scroll bars. Tricks must be employed to create them. (See "Dueling Scroll Fields" in *The Waite Group's Tricks of the HyperTalk Masters* for an example.) External commands are also available that allow scroll bars to be added to HyperCard.

HyperCard Breaks from Macintosh Interface
HyperCard breaks away from traditions established for the Macintosh interface. These may give you or your stack users trouble. The main divergence is that HyperCard buttons only use a single click and cards can't be resized like regular Macintosh windows.

Dialog and Alerts

Dialog boxes are rectangular areas that pop up in response to a menu selection or clicking an icon. They are used for displaying information or program settings and soliciting user input. Dialogs are usually composed of buttons, text fields, and sometimes even scroll bars. Dialogs are usually *modal*. When they appear, you can't ac-

cess the menu system until they are put away and when you click outside them, you get a beep. Sometimes, dialogs require no response at all and are used to inform you of events. For example, when you print a card, a small dialog lets you know printing is occurring.

HyperCard uses dialog boxes to let you choose button or field characteristics, to display the statistics of the stack, and so on. It is easy to create your own custom dialog boxes in HyperTalk because all the tools are there (buttons, paint tools for creating the box, etc).

Alerts are dialog boxes that are used to provide a warning to the user. For example, errors in HyperTalk are presented in alert boxes. Usually a button control called OK is provided to let you continue after you get an alert box.

Resources

One item that sets the Macintosh apart from most computers is its use of *resources* (menus, fonts, icons, sounds, dialog boxes, and even program code itself) that are stored in a special part of a program's file called the *resource fork*. A resource may be any chunk of data that does not need to be *compiled* along with the program itself. Compiling is the boiling down of program statements from a high-level language specified by the programmer into the language of the microprocessor in the Macintosh. In the past, such data was included with the compiled code itself, so if you wanted to change the location of a dialog box or add a new font, you would recompile the entire program. This can be very time-consuming. (Bill Atkinson mentions that the source code for HyperCard took 45 minutes to compile on a Macintosh II.) It also can be dangerous because, if you make a mistake, the whole program can stop working.

Resources are easy to change if you have ResEdit, available from Apple's documentation arm, Apple Programmer's and Developer's Association (APDA). (See the last section of this Introduction, Resources.) There is also an external command called ResCopy, written by Steve Maller, which does much of what ResEdit does, and which can be added to your stack. With ResEdit, you can change anything you see on your Macintosh screen. You can move icons, change the way they look with an icon editor, change cursors, and so on. Mostly this involves cutting and pasting since there are no actual programs to compile.

HyperCard has no way to directly add and edit resources. Apparently, Bill Atkinson felt this was giving the user too much power. He did make it easy to manipulate resources if you have ResEdit. Sound, for example, is stored as a resource, so you can cut and paste various digitized sound effects easily into your stack.

Text Editing

Macintosh deviates from standard computer technology in the way it handles typing and text. A special toolbox editor called TE (text editor) provides a common base for all editing actions. The editor allows inserting new text, deleting characters with the backspace key, using the mouse to make selections in the text, automatic scrolling of text in a window, and automatic movement of the insertion point with the keyboard

arrow keys. This means that whether you are entering text in a dialog box field, changing the name of a file, or entering text in a HyperCard field, the techniques are common and nothing new needs to be learned. Text editing in HyperCard works as it does in all Macintosh applications.

Scrolling Windows, File Dialogs, Folders, and Descriptive Filenames

The Macintosh file system is quite easy to use. *Files* are indicated by distinct icons and can be viewed by icon, creation date, size, or type. The filename can be up to 31 characters long. Compare this with a maximum of 8 characters for the older IBM PC and you'll understand why Apple is doing so well in the computer market. Apple allows files to be placed in what are called *folders*, images that look like a paper folder. When you double-click a folder from the desktop, the folder opens to reveal a window with the filenames in a scrolling list. When you open a file from an application, you see the files in a scrolling dialog box, and icons are not displayed. In either case, type the first name of the file you want and it comes into view in the box.

HyperCard treats files in a standard manner. Still lacking is a standard way to give the user a scrolling file box from HyperTalk. An external command from Steve Maller gets around this.

Event-Driven Programs

Compared with conventional computer systems the Macintosh internal structures are quite different. Instead of being program-driven, Mac applications are *event-driven*, or user-driven. Macintosh applications are controlled by the user instead of by the computer program. Conventional systems solicit user input through a fixed series of prompts, but in the Macintosh, programs respond to a wide range of events—keyboard events, mouse button events, and so on. In fact, even when a window is clicked, an event is initiated. This type of programming requires considerably more design and programming work but leads to an interface that is more flexible and efficient. It also makes for less errors and more fluid responses. For example, you can click a button, insert a disk, and press a key at the same time and the Macintosh responds to each event in a smooth and logical fashion.

HyperTalk is based on an event- and message-driven technology. Don't confuse this with the internal event system of the Macintosh. Events in HyperTalk can be sent by the mouse (`mouseUp`, `mouseDown`, etc), or a key (`commandKey`), but messages are the main way that events are propagated in HyperCard. Any statement that can be run by HyperTalk is like an event—it causes a message to be sent. More about this later in this chapter.

The Thought Police (Apple User Interface Standard)

The Macintosh is built so that all programs look and work in similar ways and learning one application makes it easy to learn another. To enforce these rules, Apple has

developed a book called *Human Interface Guidelines: The Apple Desktop Interface*. If you are serious about writing HyperCard stacks, get this book and study it carefully. You will find all the years of Apple research regarding how people best use and learn computers, and this information leads to better stacks that more people use. (See the last section of this Introduction, Resources.)

Now that you have an overview of what makes the Macintosh different from conventional computers, let's take a look at what makes HyperTalk different from conventional computer programming languages.

The Nature of HyperTalk

HyperTalk as a language has its roots in languages like Lisp and SmallTalk. These were the first languages to put forth ideas of object programming as a way to give more power to the programmer.

Computer languages can be viewed very much like complex plans for building a boat or cooking a turkey. The cook's job is to specify what ingredients to use in the recipe, in what order to use them, the settings of the oven, and so on. Similarly, it is the programmer's job to write a set of directions for the computer to follow so that it does things that are useful, such as act like a word processor, a finder, or a desk accessory. In the programmer's world, this set of instructions is called the *language vocabulary*.

The Problem with Traditional Languages

There is a problem with these computer languages, however. When the program gets complicated, it is very difficult to make it work correctly. It becomes a long, linear sequence of bizarre instructions that only a few people can understand, like a long recipe that involves hundreds of different kinds of fish, fowl, vegetables, and spices, is written in Slalovian, and includes finding the head of a toad. These programs get particularly unwieldy when the programmer wishes to use modern interface methods, such as windows, menus, scrolling text, buttons, and dialog boxes. Although these features seem simple on the outside, the coding underneath is absurdly complex and difficult. HyperTalk offers three things that set it apart from these complex linear languages.

Englishlike Syntax

First, it uses a *syntax* (vocabulary) that is astonishingly like real English. Indeed, HyperTalk has a "grammar" that allows you to write programs that can be easily understood the first time they are read. Languages like BASIC require much more elaborate statements to do such a simple task. The following program example demonstrates word switching within a sentence using HyperTalk, BASIC, and C.

HyperTalk

```
on mouseUp
  put "My favorite number is one" into sentence
  put the last word of sentence before the first word of sentence
end mouseUP
```

BASIC

```
LET A$ = "My favorite number is one"
LET L = LEN(A$)
REM FIND WHERE THE LAST WORD STARTS
REM BY LOOKING FOR A SPACE CHARACTER
FOR I = L TO 1 STEP -1
IF RIGHT$(A$,I) = " " THEN
      REM WHEN SPACE FOUND ASSEMBLE THE LAST WORD
      FOR J = I TO L
     B$ = B$ + RIGHT$(A$,J)
   NEXT J
END IF
NEXT I
REM NOW PUT THE WORD IN FRONT OF FIRST WORD
A$ = B$ + A$
```

C Function

```
/* Function last2first */

#include <stdio.h>
#include <string.h>
void last2first(char *string);   /* Prototype */
main()
{
    char input[80];
    printf("Enter a sentence: ");
    gets(input);
/* Put last word in front of first and print result */
    last2first(input);
    print("Result = %s\n", input);
}
/*-------------------------------------------------------------*/
/* This routine extracts the last word from a sentence and
 * puts it in front. The words are assumed to be separated
 * by spaces. The operation is performed in place without
 * duplicating the input string.
 */
void last2first(char *string)
{
    char *last_word, *p_start, *p_c;
/* Search for a space from the end of the line */
    if((last_word = strrchr(string, ' ')) == NULL) return;
/* Return because the last word is also the only word */
    p_start = last_word;
/* Otherwise, first skip the space */
    last_word++;
/* Shuffle characters around */
```

continued

```
    for( ; *last_word != '\0'; p_start++)
    {
        for(p_c = p_start; p_c != string; *p_c =
*(p_c-1), p_c--)
            ;  /* null statement */
        *string++ = *last_word++;
    }
/* Put a space after the word */
    for(p_c = p_start; p_c != string; *p_c =
*(p_c-1), p_c--)
        ;  /* null statement */
    *string = ' ';
}
```

Here, both BASIC and HyperTalk take the last word of a sentence and place it before the first word. Note how much easier HyperTalk is to understand and how few lines it needs to accomplish the task. Note how cryptic C is compared with HyperTalk.

HyperTalk also contains action verbs like `mouseDown` which HyperTalk calls *system messages*, and nouns like "button" which HyperTalk calls *objects*. There are easy-to-use connective verbs like `is` that serve as *operators* so that you can say things like `is in`. HyperTalk has articles like `the` and prepositions like `into` so that you can say things like `put the last word into holder`. In fact, HyperTalk is so much friendlier than other computer languages that it doesn't even use the word "program." In HyperTalk, a set of statements that performs some task is called a *script*. A script is much like a subroutine in BASIC or a procedure in Pascal. It is a snippet of code. Unlike subroutines and procedures, HyperTalk scripts are not "called" by a main program, but come to life whenever a message is sent to them.

Well-written HyperTalk scripts are much easier to understand because the verbs and nouns are so descriptive. In some cases, HyperTalk is able to interpret differently phrased statements as meaning the same thing. All the following HyperTalk statements mean the same thing and can be used interchangeably: `go to card 1`, `go to first card`, `go card one`, `go first`.. If you are interested in comparisons between Pascal and HyperTalk, see the *Macintosh HyperCard Script Language Guide*.

Even with this Englishlike grammar, writing HyperTalk scripts requires a knowledge of the HyperTalk vocabulary and the rules that govern how HyperTalk words are put together.

Object-Oriented Programming

Another feature of HyperTalk that sets it apart from conventional languages is that it uses objects and techniques of *object-oriented programming*. You will be hearing a lot about objects in the coming years. In a loose sense, this means that HyperTalk is based on a set of software objects or parts that are created in HyperCard. These are elements like buttons, fields, cards, and so on. These objects can be created manually from the HyperCard menu system and, as an author, you may have done so. Remember, objects are characterized by special dialog boxes that allow you to control their characteristics. These characteristics are called *properties* of the objects.

For example, a text field can be set to many styles, fonts, and shapes, and can be positioned on the screen. It can be made invisible or visible. These properties are somewhat like variables, except that they don't lose their values when you turn off the program.

Most computer languages are *procedural*, meaning they follow a sequence of steps starting at a certain place and ending at another place. HyperTalk is not procedural and forces new kinds of thinking about programming. We will be learning a lot about objects in this book, starting in chapter 2, "Objects."

New Things to Learn

HyperTalk has many similarities to Pascal. It provides sophisticated control structures, user-defined subroutines and functions with parameter passing, string manipulation functions, a variety of data structures, text display, graphics, and sound commands, as well as disk file routines. Its file and printing facilities are rather weak; there are no random access files and printing doesn't always work as you might wish. Pascal programmers should have no trouble learning how to use HyperTalk. BASIC programmers will love its simplified string handling. However, there will be a learning curve to climb over in the area of objects and messages.

Inheritance and Properties

Along with HyperTalk's object-oriented programming comes a feature called *inheritance*. When you create an object in HyperCard, it comes with a host of built-in traits, as we mentioned, called properties. These properties stay around when you leave the stack, and don't change unless you change them on purpose. One of the most interesting features of objects is that they can be copied to the clipboard, just like text and graphics, and then pasted into a different location or different stack. Even more interesting is that when the object is copied, all the traits it had, including its script, come along with it. This makes it easy to build interfaces because you can, for example, make a special button object that has a certain script and style, and then repeatedly clone it by copying and pasting it. This is the inheritance feature—we will be showing you how to exploit it throughout this book.

Message-Driven Programming

The third feature that sets HyperTalk apart from other languages is that it uses *messages*. A message can come from the mouse or the keyboard as well as other places, as we shall see. The messages have simple names like `mouseUp`, `mouseStillDown`, `openCard`, `idle`, `startup`, `quit`, and so on. Scripts are designed to react to, or handle, events that are sent to them in the form of messages. A script inside an object is referred to as a *handler*. Handlers, as we will show, handle messages. For example, a `mouseUp` message is sent to an object like a button. If the button contains a handler that begins with `on mouseUp`, and ends with `end mouseUp`, it will "trap" the `mouseUp` message and its script will execute.

This idea of sending messages to little program snippets living inside fields, buttons, cards, and other objects may seem quite strange at first. But as you delve more into using HyperTalk, it becomes quite natural.

Zenlike Nature

Another unique element of HyperTalk is that the objects created become holders for the scripts, which, taken as a group, make up a HyperTalk program or stack. Thus, the functionality of a stack is spread throughout the buttons, fields, cards, backgrounds, and, as we shall see toward the end of the book, even into the external commands that may be attached to the stack.

This is very unlike the case of the traditional, nonobject-oriented, linear language we described above, in which the program resides in a single long file of a certain name. To get a complete description of such a program, you must scan it with a word processor and occasionally jump to included files.

But in a Zenlike manner, a HyperTalk program is in all places at the same time. And indeed a HyperTalk script can't be fully comprehended without examining many, if not all, of the objects in the stack. (Some of the first shareware third-party stacks were script reporters that made up for HyperTalk's rather weak editor, as we shall see.)

True, this is not a book on the history of computer programming or improving HyperCard's reporting features, but the point is important: HyperTalk offers an object-environment that is unlike any programming language. This makes it both a more powerful environment and more of a challenge to master.

Hyper Facts

In the HyperTalk Garden Da Vita

For those who want the whole story right away, here it is: HyperTalk is synergistic, user-friendly, and fully organic. It permanently alters the way we teach computers and computer programming.

In the early days of electronics, computers—and, indeed, most electronic gadgets—were made of individual parts: transistors, resistors, capacitors, coils of wire, and so on. If you wanted a radio, you made it with hundreds of these parts. A television took thousands and a computer took hundreds of thousands. Someone had the bright idea of taking a huge number of transistors, putting them into a small plastic package with pins on it, and making up some standard functional components that a designer could tune with a few outboard parts, thus giving the functional equivalent of much larger and more complex components. This tiny package of functions was called an *integrated circuit* or a *chip*, and it led to giant steps in technology because more and more devices could be miniaturized. The Macintosh is full of such components. Indeed, the Macintosh represents state of the art in integrated and super-integrated components used in a consumer computer.

Now consider programming languages. They are supposedly the tools you use to build application programs that run on the computer to solve basic problems. After you play with some of these languages, you begin to see that they are actually like the old transistor components of

the past. In order to do anything substantial, you must build every functional part of your program completely from scratch using single lines of program instructions. (Much of this programming also involves moving tiny bits of abstract data.) Programs of any substance— commercial word processors, painting programs, database systems—offer the user an incredible host of features, such as windows, menus, dialog boxes, drawing tools, and so on. However, these features are made of thousands of lines of assembly code (software transistors) and take hundreds of hours to devise and maintain. And once built, these structures are only one particular person's idea, not a standard that others can use.

You can lump almost every popular computer language sold today—BASIC, C, Assembler, Fortran, Cobol, Forth, Lisp, etc.—into the transistor language category.

Depending on what end of the tunnel you look down, HyperCard gives a host of standard objects found in the normal computer interface—fields, dialogs, buttons, etc.—the ability to be completely manipulated from a high-level, Englishlike computer language as if the objects were parts of an erector set. That may be a lot to swallow, but it simply means that HyperTalk diverges from general-purpose programming languages in a vital manner.

HyperTalk is an object-oriented language. Its objects have the capability of inheriting traits when duplicated, and those traits include the style of the objects, fonts they contain, locations on the screen, and even the programs residing inside these objects.

What does all this have to do with learning to program? Well, transistor languages like BASIC force you to learn to program by writing dozens of lines of code to accomplish the most primitive tasks—printing columns of text on the screen, looping while waiting for keyboard input, moving a text character to a new location. Anything as simple as moving a window, pulling down a menu, changing a cursor, scrolling a field, or drawing a graphic requires an astonishing number of lines of code. HyperTalk allows manipulation of very complex software objects, using incredibly simple statements. HyperTalk programs are written in a completely different way from the transistor languages. Transistor languages contain a long printout of instructions. HyperTalk scripts are spread out in multiple objects.

In addition to its objects, HyperTalk is based on a wonderful idea: a message-driven interface. Devices like the mouse, as well as certain system events, send messages to HyperCard objects. Handlers inside these objects react to the messages and come alive if the message they are waiting for is detected. This adds an organic nature to HyperTalk programming.

HyperCard Tricks for Better Scripts

Using HyperCard's scripting level requires experience with the lower user levels: browsing, writing, painting, and authoring. Although many of you may already have

mastered these levels, there are a host of shortcuts and tricks that will make your scripting experience much more effective. You may use keyboard substitutes for the menus, navigate between cards, open scripts easily in the various HyperCard objects, and so on. Learning these tricks now can save a great deal of time when you are writing scripts.

Keyboard Alternatives

HyperCard offers several built-in keyboard alternatives to commonly used menu items. For example, instead of pulling down a menu each time you wish to make a new card, you can simply type Command-M from the keyboard. This saves time because you don't have to do any mouse movements (point and click). A new stack can be created with `Open Stack` (Command-O). You can switch between the background and foreground modes by typing `Command-B`. The *Message box windoid* (a small window that you can move around and send away but not resize) can be shown and hidden with Command-M. Buttons and fields can be moved forward with Command-= or sent further back with Command[—] (Command-minus). You can quit HyperCard with Command-Q.

All of the navigation commands have keyboard alternatives you should learn. `go first card` is Command-1 and `go last card` is Command-4. `go previous` is Command-2 and `go next` is Command-3. (Note how symmetrically Bill Atkinson has laid these keys out.) Don't forget that all the Clipboard keyboard alternatives are operational in HyperCard, and work for objects as well as text. Paste with Command-V, copy with Command-C, and cut with Command-X.

The Select menu command is one of the nicest of HyperCard's painting features because it lets you automatically select the last graphic element created. The Command-S keyboard alternative will make you wonder how you ever used MacPaint without it.

Beautiful Tools

We would like to make an observation about HyperCard. It seems that most of its tools, keyboard alternatives, power keys, and ways of doing things have an extraordinary degree of symmetry and consistency not found in most software today. After using HyperCard for some months you discover that there is an amazing degree of precision and interactivity between the various levels. For example, when we began writing this book, we decided to use one of the more powerful commercial paint programs for doing our artwork. As time went on, we found that Bill Atkinson's keyboard substitutes for the paint tools were so good that it was much faster to use HyperCard's built-in paint tools. Occasionally, HyperCard's paint tools fall down, and we have to switch, but in general, they are far superior to most programs. You can rotate graphics, fill black paint with a pattern, and create other exotic graphics. We recommend you spend time using them.

Power Keys

There are quite a number of *power keys* that let you perform Painting menu items with a single keystroke. For these to work, you must have the paint tool selected. For example, after you make a selection, typing `F` for Fill lets you fill that selection with the current pattern. Likewise, after you select a graphic in the paint mode, you can type `H` for Flip Horizontal, `I` for Invert (which inverts the current selection), and `D` for Darken (which scatters pixels across a selection).

For a complete discussion of these power keys, see the *HyperCard User Guide*. You might want to tear out the reference card, if you haven't already, and put it next to your computer.

Keyboard Manipulation

The keyboard also comes in handy when you are manipulating buttons and fields. If you select the field tool and drag a field with the option key pressed, you create a clone of the original field. The new field inherits all the original field's properties (its style, the text characteristics, etc.) but not the actual text contents (if any) of the field.

If you hold the Shift key down with the option key while you drag to the left or right, the new field or button moves along a fixed horizontal or vertical line (depending on which way you start moving). This is great when you want to create a row or column of buttons all on the same horizontal or vertical line.

Windoid Controls

You probably know that you can drag the Tool and Pattern menu palettes off the menu bar and place them on the screen. These little tear-off menus are actually windoids with a drag bar and a close box just like their bigger relatives. The Message box is also a windoid and, just as you can make it appear and disappear with Command-M, you can toggle the Tool palette on and off with Command-Tab and the Pattern palette (when a painting tool is active) with a simple Tab. Make the menu bar appear and disappear with Command-Space.

Fast Info

To quickly open the button info dialog box for a button *when the button tool is selected*, simply double-click the button you are interested in. If you press the Shift key as you double-click, you open the script editor rather than the button info dialog box displaying the script in the button.

You get the field info dialog box the same way. Just select the field tool and double-click the field. You can even open the field's script by holding down the Shift key, though field scripts are usually empty.

Text Selections and the Clipboard

As with any other good Macintosh application, HyperCard lets you select a word in a text field by double-clicking it and lets you select any combination of letters or words by dragging through them with the mouse.

Once you've selected something, use the keyboard alternatives mentioned previously to put the selection onto the Clipboard with a Command-C (copy) or a Command-X (cut). To insert the contents of the Clipboard after the insertion point cursor, press Command-V (paste). To undo the most recent Clipboard operation, press Command-Z (undo).

New Shortcuts

Although you could select the browse tool by pressing Command-Tab, now you can also select the button tool with Command-Tab-Tab (press the Tab key twice, quickly) and you can select the field tool with Command-Tab-Tab-Tab (press the Tab key three times, quickly). As these are probably the most frequently used tools, at least for the author and scriptor, this new access really comes in handy.

Something New: Peeking

You can now quickly peek at any script with the following keyboard-mouse combinations: the stack script, option-Command-S; the current background script, option-Command-B; the current card script, option-Command-C; visible buttons, option-Command-Click (with the browse or button tool); hidden buttons, option-Command-Click (with the button tool); visible fields, option-Command-Shift-Click (with the browse tool or the field tool); or hidden fields, option-Command-Click (with the field tool).

Resources

This section presents names and addresses of on-line, public-access computer networks that have a large HyperCard and HyperTalk following. We also list other related sources for you to tap.

Official HyperCard Utilities

The Apple Programmer's and Developer's Association (APDA) publishes a quarterly catalog of Apple programming languages, development systems, utilities, and tech notes, and has over 21,000 members who are commercial hardware or software developers. APDA is a division of A.P.P.L.E., with offices at 20525 Mariani Avenue, M/S

33G, Cupertino, California, 95014-6299. APDA membership dues are $20 per year ($25 in Canada and Mexico, $35 elsewhere).

Networks

Compuserve and The Well are on-line public access computer networks with large HyperCard and HyperTalk followings.

Compuserve

Compuserve has the most HyperTalk and HyperCard activity. There is a special forum called AppHyper that contains thousands of messages from users seeking advice. There are also data libraries with thousands of HyperTalk Stacks. All the HyperTalk gurus "hang out" on Compuserve, including Winkler, Shafer, Goodman, and so on. Compuserve can be reached at

Service: Compuserve
Subscriber Info: (614) 457-8600
Baud Rate: 300 or 1200 baud
Hours: 24 hours per day

You can contact The Waite Group via Compuserve as follows: author, Mitchell Waite; user ID, 74146,3515.

The Well

The Well is a "regional" computer network system in Sausalito, California, near The Waite Group's offices. This is a conference system put together by Stewart Brand, creator and editor of the *Whole Earth Catalog* and *Whole Earth* magazine. The system is a public access UNIX system based on a large VAX computer. The UNIX is hidden by a friendly conferencing shell called pico span. We mention this system because there is a very large following of HyperCard and HyperTalk users and some of the most talented gurus frequent this system, including the authors of this book. You can subscribe to The Well by calling from your computer via a modem and following the "new user" instructions:

Service: The Well
Modem Line: (415) 332-6106
Subscriber Line: (415) 332-4335
Baud Rate: 300 or 1200 baud
Hours: 24 hours per day

The authors of this book can be reached by sending electronic mail to the following user names:

Mitchell Waite: mitch
Stephen Prata: unicom!stephen
Ted Jones: theo

Magazines

The following are magazines devoted to HyperTalk and HyperCard. In addition, we recommend the article "Comparing HyperTalk to Pascal" by Paul F. Merrill in *MacTutor, The Macintosh Programming Journal* (vol. 4, no. 9, September 1988, page 14) for a comparison of the HyperTalk language with Pascal.

Hands On Macintosh

Formerly *Nibble Mac*, this is the magazine for Macintosh enthusiasts. Available from MicroSparc, Inc. (52 Domino Drive, Concord, Massachusetts, 01742, (508) 371-1660), this is a mass-circulation periodical devoted to HyperTalk scripting. Published eight times a year, it also has a large Macintosh BASIC coverage. An author of this book has a column in *Hands On* called "Mitchell Waite's HyperTalk Tricks."

HyperLink

This small subscription magazine has strictly HyperTalk coverage. Some of the scripts may seem elementary but, in general, there is a high degree of professional care devoted to this magazine.

Books

Rather than bore you with the standard list of HyperCard and HyperTalk books, we provide a list with analysis of the current books on the market. Table I-1 presents the nine most popular HyperTalk books at the time of this writing and Table I-2 covers five popular HyperCard books.

The HyperTalk books in Table I-1 are arranged in order of pages devoted to teaching HyperTalk using tutorials versus pages devoted to reference. Books with more pages on HyperTalk reference are toward the end of the table and the ones with the most tutorial on HyperTalk are toward the beginning.

Many of the books devote a significant number of pages to referencing the HyperTalk language, now covered in a separate manual (Addison-Wesley 1988), because Apple brought out HyperCard as a product before a manual for the HyperTalk language was ready. There was a strong demand for such information, and Danny Goodman, who worked with Apple, published a book that contained a good deal of HyperTalk reference material. This was repeated in subsequent books from other publishers that were brought out in 1988. In late 1988, Apple's manual made it unnecessary to include such material.

Table I-2 ranks the books by number of pages devoted to HyperCard, subtracting pages on HyperTalk. The books with the most HyperCard pages are at the beginning of the table.

Table I-1. HyperTALK and XCMD Books

	The Waite Group's HyperTalk Bible	HyperTalk Programming, 2nd Ed.	Mastering HyperTalk
Price	$26.95	$24.95	$24.95
ISBN		0-672-48426-9	0-471-61593-5
HyperCard, HyperTalk, XCMDs	HyperTalk	HyperTalk	HyperTalk
Publisher	Hayden Books/Sams	Hayden Books/Sams	John Wiley & Sons
Author(s)	Mitchell Waite, S. Prata, & T. Jones	Dan Shafer	Keith Weiskamp and Namir Shammas
Publication Date	1989	early 1988	late 1988
Total Pages	600	546	506
Pages of HyperTalk Reference (2)	0	23	31
Pages of HyperTalk Teaching (3)	**600**	**523**	**475**
Prog. Listing Disk and Cost	$19.95	$20.00	$24.95
Covers 1.2	Yes	Yes	Yes
Covers International 1.2.2	Yes	No	No
Ref. Card or Poster Included	No	Yes	No

	Running HyperCard with HyperTalk	Dr. Dobb's Essential HyperTalk Handbook	Apple HyperCard Script Language Guide: The HyperTalk Language
Price	19.95	$24.95	$22.95
ISBN	0-943518-79-2	0-934375-98-4	0-201-17632-7
HyperCard, HyperTalk, XCMDs	Both (1)	HyperTalk	HyperTalk
Publisher	MIS Press	M & T Publishing, Inc.	Addison-Wesley
Author(s)	Barry Shell	Michael Swaine	none
Publication Date	late 1988	late 1988	late 1988
Total Pages	563	511	317
Pages of HyperTalk Reference (2)	156	280	154
Pages of HyperTalk Teaching (3)	**407**	**231**	**163**
Prog. Listing Disk and Cost	$20.00	$20.00	$25.00
Covers 1.2	Yes	Yes	Yes
Covers International 1.2.2	No	No	No
Ref. Card or Poster Included	No	No	No

	Concise Guide to HyperTalk	**XCMDs for HyperTalk**
Price	$9.95	$24.95
ISBN	0-943518-84-9	0-943518-85-7
HyperCard, HyperTalk, XCMDs	HyperTalk	XCMDs
Publisher	MIS Press	MIS Press
Author(s)	Barry Shell	Gary Bond
Publication Date	early 1988	late 1988
Total Pages	166	468
Pages of HyperTalk Reference (2)	166	NA
Pages of HyperTalk Teaching (3)	**0**	**NA**
Prog. Listing Disk and Cost	No	$25.00
Covers 1.2	No	Yes
Covers International 1.2.2	No	No
Ref. Card or Poster Included	No	No

1. This book claims to teach HyperTalk while it teaches HyperCard.
2. Until recently, Apple did not have a reference manual for HyperTalk. Therefore, many of these books waste various amounts of pages repeating what is in the Apple reference manual.
3. This figure represents the number of pages devoted to teaching HyperTalk and is a figure of merit for gauging the book.

Table I-2. HyperCARD Books

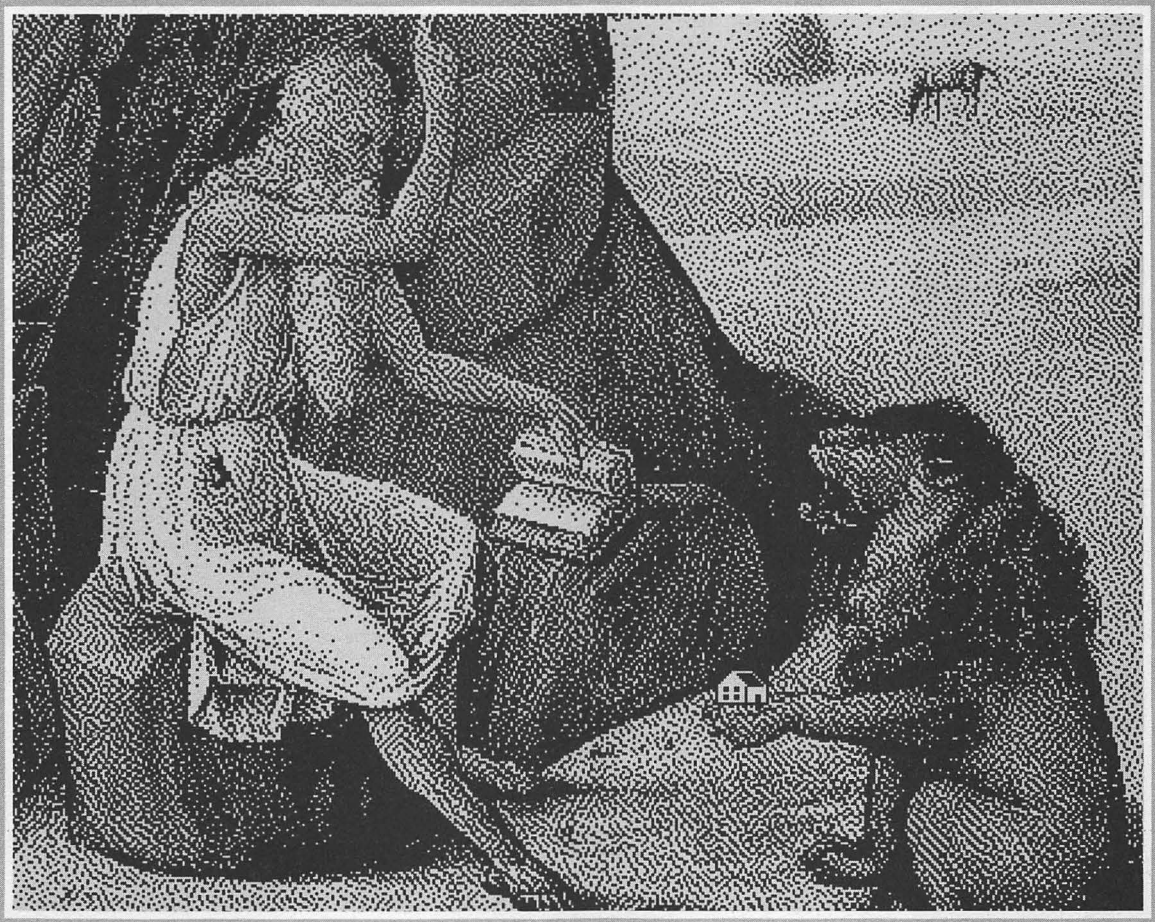
	The Complete HyperCard Handbook	Danny Goodman's HyperCard Developer's Guide	Using HyperCard Home to HyperTalk
Price	$29.95	$24.95	$24.95
ISBN	0-553-34391-2>2995	0-553-34576-1>2495	0-88022-340-5
HyperCard, HyperTalk, XCMDs	HyperCard (1)	HyperCard (2)	HyperCard
Publisher	Bantam Computer Books	Bantam Computer Books	Que
Author(s)	Danny Goodman	Danny Goodman	Vaughan, et al.
Publication Date	late 1987	late 1988	late 1988
Total Pages	721	644	750
Pages of HyperTalk Pages	190	200	363
Pages of HyperCard Teaching	**531**	**444**	**387**
Prog. Listing Disk and Cost	No	$13.95	$29.95
Covers 1.2	Yes	Yes	Yes

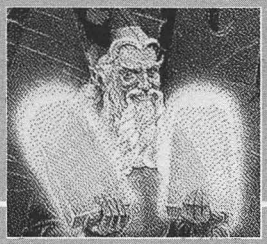
	Understanding HyperCard	HyperCard Made Easy
Price	$24.95	$19.95
ISBN	0-89588-506-9	0-673-38358-X
HyperCard, HyperTalk, XCMDs	HyperCard	HyperCard
Publisher	Sybex	Scott, Foresman & Co.
Author(s)	Greg Harvey	Dr. William Sanders
Publication Date	mid 1988	mid 1988
Total Pages	581	377
Pages of HyperTalk Pages	276	230
Pages of HyperCard Teaching	**305**	**147**
Prog. Listing Disk and Cost	No	No
Covers 1.2	No	No

1. This book was the first book on HyperCard. It has a rough HyperTalk section.
2. This book is a strange mix of issues relevant to developers.

Chapter 1

Starting





Quick View

Commands

`beep`
`put`
`set`

Concepts

Strategy
Version 1.1 or 1.2?
Preferences first
Making your practice stack
The `script` property
Your first script

Concepts *cont'd*

Case rules
Using the script editor
Automatic formatting
Option-Return for long lines
HyperCard-HyperTalk linking
Comments in scripts
The Message box
HyperTalk's vocabulary
Our syntax
`put` and `set`

Chapter 1

Starting

Here is a good strategy for learning anything complex, be it writing a computer program, baking an upside-down cake, or building a septic tank: Quickly run though the entire process, skimming its primary subjects, leaping over its more abstract and difficult topics. Next, repeat this process, spending more time on each important idea and dealing with the abstract topics in more detail. Do exercises along the way. Do this process again and spend more time with each topic. In a nutshell: use a spiral approach to get the big picture, then circle around the topics until they are understood completely.

This book is constructed with this spiral approach. This chapter gives you the preliminaries needed to use scripting tools that HyperTalk provides—the script editor and the Message box. We introduce your first script (in a button), show you some navigation, and explain two often-used commands: `put` and `set`. We tell you just enough about objects and messages to do something useful with them. Don't worry if everything is not crystal clear this time around. By the end of this chapter, you should have a good overview of how HyperTalk scripts are built and how some of HyperCard's objects work.

What Version of HyperCard?

This book is compatible with all versions of HyperCard, including, at the time of this writing: 1.0.1, 1.1, 1.2, and 1.2.2. Whenever a feature is specific to a particular version (usually 1.2 only, for example) we present an icon alerting you to that fact. We suggest you update your version of HyperCard to 1.2, because that version has features that are extremely useful while you are developing scripts.

User Level and Preferences

Before you start learning to script, you must make sure that your HyperCard is set to the correct *user level*. HyperCard provides five levels of user involvement. These

levels are controlled by the last card in the Home stack, the User Preferences card (see Figure 1-1).

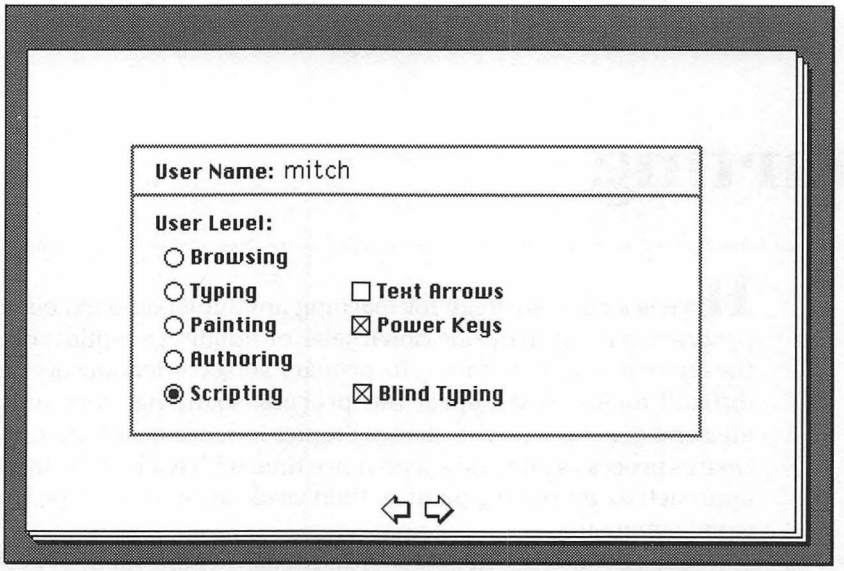


Figure 1-1. User Preferences settings.

The five levels are controlled by choosing one of the five buttons on this card. Each higher level of control gives users access to additional tools, and each level contains the abilities of the levels lower than it. The five levels are:

1. *Browsing.* At the browsing level, you can only look at the information in the stack. You can't change anything or copy any information. This corresponds to conventional read-only access. You will find lots of commercial stack products set to this user level.
2. *Typing.* At the typing level, you can enter information into the stack and copy or change text. You are given read-write access. However, you can't alter any of the structure of the stack or change its objects in any way. If your stack is to be used by nonscriptors and you want them to store information or copy information, this is the level to use.
3. *Painting.* At this level, you can add and copy existing graphics with HyperCard's painting tools. You can change the appearance, but not the functionality. If you want users to be able to import or export graphics and text from your stack, this is the level to use.
4. *Authoring.* At the authoring level, you can modify fields and buttons in the stack, make new backgrounds, add cards, modify graphics, edit or add text, and generally change everything except the scripts in the stack. This level is used when you want to give users the most freedom, but don't want them to copy or modify the scripts. Perhaps your scripts contain competitive advan-

tages you don't wish to give away, or you want to prevent the user from harming the functioning of the stack by changing a script.

5. *Scripting.* The scripting level is the highest user level. You have complete control over the stack, including the ability to open any of the scripts and modify them. This is the level that most users like because they have the most freedom to use the features of your stack and to learn from it. For example, a customized button that clicks and changes icons to appear to close and open is useless if you can't access its script.

Set Preferences to Scripting
All examples in this book assume that you have set the user level to level 5, the scripting user level. Set this level from the Preferences card, the last card of your Home stack.

Figure 1-1 also shows three check boxes labeled Text Arrows, Power Keys, and Blind Typing. When you check Text Arrows, the text arrows on the keyboard move the cursor between text characters in a field rather than between cards in the stack. Leave it unchecked for now. Checking Power Keys enables you to use the keyboard to simulate certain menu commands. This is a helpful feature while you script, so check it. Checking Blind Typing enables you to type into the *Message box*, a small *windoid* for trying out HyperTalk commands even when the Message box is hidden from view. (A windoid is a window that you can move around and make go away, but not resize.) Check this box, too.

These levels can also be set from within a script (we will learn what that is soon enough), so you are able to programmatically control the user level settings when your scripts start.

Test Stack: HyperTalk Bible

Before you can begin scripting, you need a stack to script in. For this book, we create a new stack and call it the HyperTalk Bible stack. It will hold all the cards, buttons, fields, graphics, and stack projects you make as you read this book. (If you want to purchase the stack already made, so that you can avoid typing and painting, see the Preface.) After we make the stack, we will make a new card.

In this first example, we guide you step by step through the creation of the stack. In later chapters, we assume you have already created the stack.

Example 1:

1. Start HyperCard by double-clicking its icon, or if it is already running, type Command-H to go to the Home stack.
2. Select Last from the Go menu to go to the last card in the Home stack. You can also press the Left Arrow key to go from the first card to the last, or you can press Command-4 to go directly to the last card of the Home stack.
3. Click the Scripting radio button. You now have enabled the HyperCard's scripting mode.

4. Click the Power Keys and Blind Typing check boxes to choose them unless they are already checked.
5. Choose New Stack... from the File menu to create a new stack.
6. Deselect `Copy current background` in the New Stack dialog box and enter `HyperTalk Bible` into the name field. By deselecting `Copy current background`, the new stack has no background graphics and doesn't look like the Home stack.
7. Click the New button.

Result:

An empty stack called HyperTalk Bible with a pure white background and one card (see Figure 1-2).

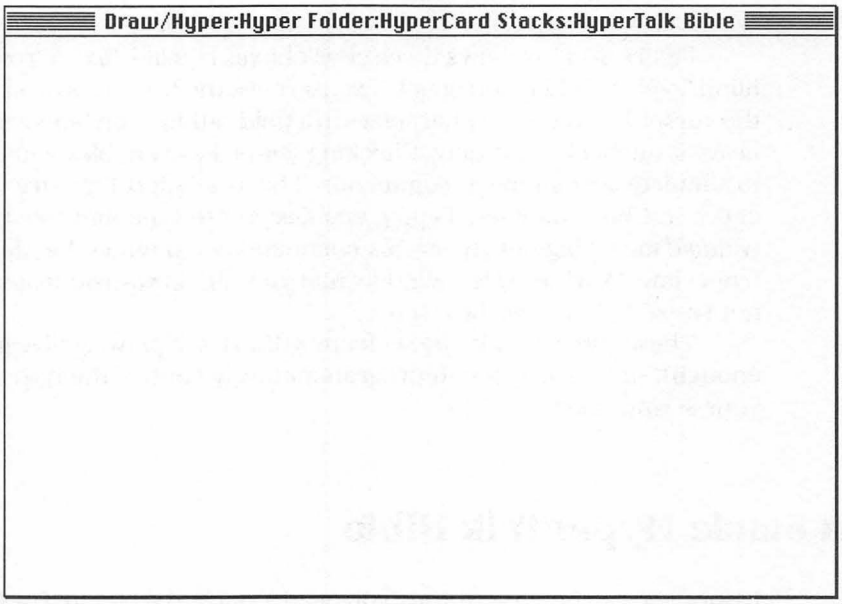
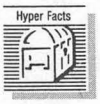


Figure 1-2. The HyperTalk Bible stack.

8. Select Card Info... from the Objects menu and enter `Getting Started` into the card name field.

Result:

You have now named the single card in the HyperTalk Bible stack `Getting Started`.



Names and Locations of Stacks

Our new stack is automatically saved by HyperCard in the same folder and location as the Home stack. Normally the Home stack is inside a folder called HyperCard Stacks, but it may also be in a different folder depending on how your computer is set up. You will find in this book that the names and locations of the stacks may change from one chapter to an-

other, and be different from your names. For example, the name of your main floppy disk or hard disk holding HyperCard appears in the top title area of any cards, and most likely is different from ours. This usually won't cause any problems, but we will warn you when it might.

The *script* Property and the Script Editor

The various objects that make up HyperCard have characteristics called *properties.* Programming is unusual in HyperTalk because of the sheer number of object properties you must be familiar with. Each object contains dozens of possible properties and you'll spend a lot of time controlling and manipulating them. It is important for you to be familiar early with HyperTalk properties.

The most important property in this book is the `script` property. Think of an object as a businessperson, and a script as a business plan. The script tells the object what to do when it is activated. The script is made up of statements built from a vocabulary of words that HyperCard understands. These words are the building blocks of the HyperTalk language. Next, we learn how to create a simple script inside a button object and also become familiar with the script editor.

Your First HyperTalk Script

Example 2:

1. Open your HyperTalk Bible stack.
2. On the blank white card make a new button (using New Button in the Objects menu) and open the button info dialog box. Figure 1-3 shows how this looks on the card.
3. Click the Script button in the lower-left corner. (In the future, you can double-click the button and hold down the Shift key to accomplish the same results, but the button tool must be selected first. In Version 1.2, you can hold the command and option keys down and just click the button once to open the script of a button.)

Result:

HyperCard's script editor dialog box opens, revealing the script of the new button (see Figure 1-4).

A *script editor* is just what its name implies. It allows you to edit the contents of the script, regardless of what object that script lives in. Every object can contain a script, and hence, every object can be examined with the script editor. Unfortunately, in the interest of getting a product to market fast, Apple left HyperTalk with a slightly deficient editor. Instead of operating like a word processor (such as MacWrite or Microsoft Word), the HyperCard script editor is more like a desk accessory editor (such as NotePad or MockWrite). It lacks abilities we have come to expect (like true search and replace, undo, variable fonts, styles, justification, rulers, and so on).

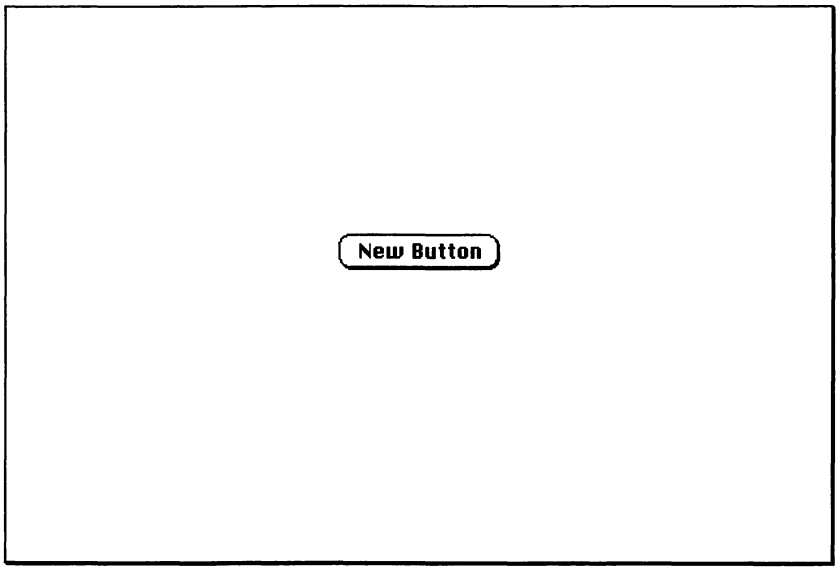


Figure 1-3. The HyperTalk Bible stack showing New Button.

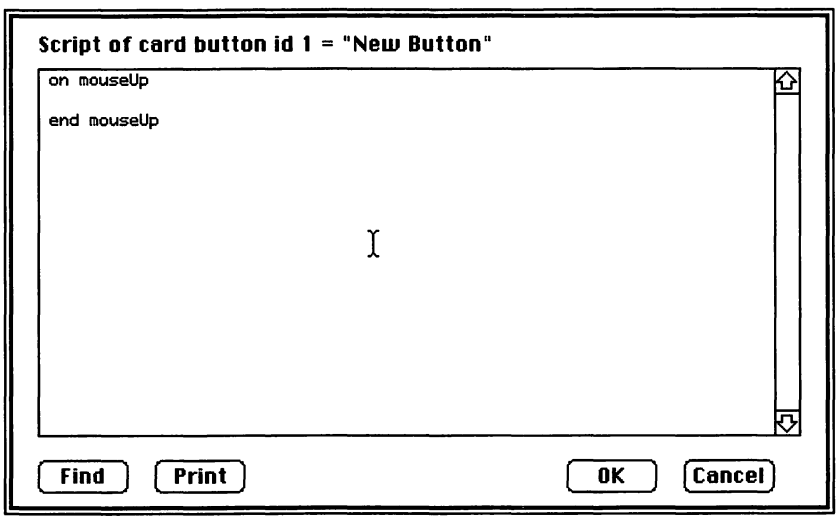


Figure 1-4. The script editor showing a handler outline.

In Figure 1-4, the editor has a button to close the script (OK), cancel it without accepting changes (Cancel), find text (Find), and print the script (Print).

When you open the HyperTalk script editor, the text shown in the dialog box

window depends on what object it was opened for. In the case of a new button, you see `on mouseUp` and `end mouseUp` framing a blank line.

These words are set up because the makers of HyperCard wanted to get you started with the basic outline of a *handler*. A handler is a small routine that starts with the statement `on something` and ends with the statement `end something`. The something can be `mouseUp`, `openCard`, or other words. HyperTalk commands are between the opening and ending statements.

When you click the mouse in HyperCard, a special message is sent to any object under it. When you press the button down, a `mouseDown` message is sent and when you release the button, a `mouseUp` message is sent. Messages are the key to understanding HyperCard programming. In the case of your button object, the opening and closing statements `on mouseUp` and `end mouseUp` tell HyperCard to watch for a user's clicking over the button, and in particular, letting up the mouse button.

When that message is received, HyperCard automatically traps the `mouseUp` message and passes control to your handler inside. Right now this handler is empty—there are no instructions between the opening and closing lines—so clicking over it does not do anything. (Try it if you don't believe us.)

The *beep* Command

Let's enter a very simple HyperTalk statement into the blank line in the mouseUp handler. This statement consists of a single word from the HyperTalk vocabulary: the command `beep`, which causes the speaker to emit a standard Macintosh beep sound. Think of a command as a verb in English that, when given a subject (a noun), forms a complete sentence to HyperTalk. When such a sentence is typed into the Message box or run in a script, HyperTalk tries to interpret and carry out whatever is described by the sentence.

Example 3:

1. Open your HyperTalk Bible stack.
2. Open the script of the New Button button. (Remember, you can do this quickly by holding down the Shift key and double-clicking the button.)
3. Place the insertion point on the blank line between `on mouseUp` and `end mouseUp` (use the cursor keys to move the cursor if you wish) and type `beep`.

   Your script should look like Figure 1-5 now. If it doesn't, use the standard Macintosh editing features to correct it.
4. Click the OK button or press the Enter key.
5. Select the browse tool and then click New Button.

Result:

A genuine Macintosh beep.

Congratulations. You are now a HyperTalk scriptor. You have just written a script that reacts to a `mouseUp` event by generating the same beep sound all the commercial programs use. See how easy HyperTalk is? Reward yourself with a cup of coffee, tea, or, if you're a Californian, with an avocado-tomato-ginseng smoothie.

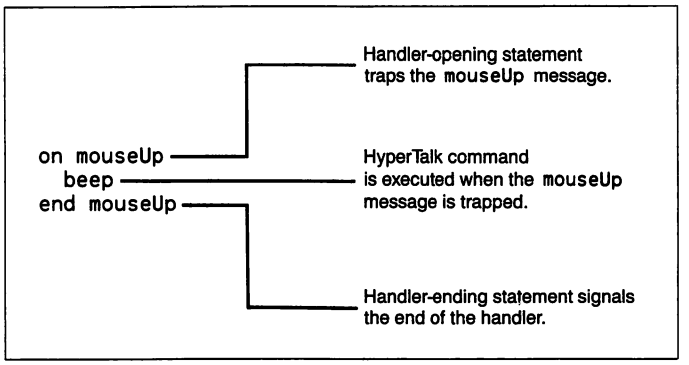


Figure 1-5. HyperTalk handler inside button.

Case Rules

Note that HyperTalk is blind to the difference between uppercase and lowercase text, or, to put it in fancier terms, HyperTalk is case-insensitive. This means that if you use `MOUSEUP` at the start of a handler and `mouseUp` at the end, HyperTalk will be none the wiser. In this book, we use upper- and lowercase to make it easier to read our scripts—`errorFlag`, `totalToDate`, and so on. You can ignore these case distinctions, but if you do, it may be harder to check your spelling and avoid errors.

Using the Script Editor

The HyperTalk editor, through Version 1.2, has little in common with more professional word processing editors like MacWrite, Microsoft Word, FullWrite, or Word Perfect (all products costing several hundred dollars). The HyperTalk script editor, scaled down from what you might expect, does what's needed. One font (Monaco), one point size (9), and one style (plain) are available. The double lines around the editor window tell us it is a Mac *modal dialog box*, like that found in the files-open and save-as dialogs. Clicking outside the dialog box causes only a beep. This means you can't drag down a menu or bring the HyperCard card window forward without first closing the script editor. This can be inconvenient, and we hope an upcoming version will correct this limitation. (Special add-on editors do exist that you can attach to HyperCard to give you more powerful script editors.)

The script editor uses the Return key for entering the carriage-return character in the script, so you can't close the dialog box by pressing the Return key. (You can close it by pressing the Enter key, however.)

Cutting, Copying, and Pasting

Although the menus are not available, you can still copy, cut, and paste but you must use the keyboard alternatives Command-C, Command-X, and Command-V. Table 1-1 shows the command keys and functions that they perform for the editor.

Table 1-1. Script Editor Keyboard Options

Key	Function
Command-A	Selects entire script
Command-C	Copies selection to the Clipboard
Command-F	Shows Find dialog box
Command-G	Goes to next occurrence of the Find text
Command-H	Copies selection into the Find field
Command-P	Prints selection or the entire script
Command-V	Pastes contents of the Clipboard to the selection
Command-X	Cuts selection to the Clipboard
Enter	Closes the script editor (same as clicking OK)
Tab	Automatically formats the script

Searching and Finding

The editor has a search capability, but no search and replace. You can find places in HyperTalk scripts fairly easily, but you can't globally change all occurrences of one word to another, as you can in word processors.

Don't Use Option Characters in Names
Macintosh has 255 special characters that can be accessed by pressing the option key with a regular keyboard key. These characters should not be used in HyperTalk for names of things, but they can be used in text you display.

Let's see how find works. Click the Find button or press Command-F to get the Find dialog box. Command-H copies selected text into the Find dialog box so that you can look for the next occurrence of it in the script. Command-G skips to the next occurrence of text already found.

Example 4:

1. Open your Home stack.
2. Open the stack script. (You can do this quickly by holding down the Shift key while selecting the Stack Info menu item.)
3. Click the Find button, enter `powerKeys`, and press the Return key.

Result:

The word `powerKeys` is selected in the GetHomeInfo handler.
The Home stack contains a large script, mainly for initializing HyperCard. We have quickly found the handler for getting information about how the user has the home card set up.

4. Select the words `card field` above `powerKeys` by dragging through them with the mouse; press the Command key and the H key to place the selected text into the Find box and to go to the next occurrence of the string.
5. Continue pressing Command-G to find more occurrences of `card field`.

Result:

All the occurrences of `card field` are found below where you selected the string. When the end of the script is reached, you will hear a beep.

Automatic Formatting of Scripts

The script editor also offers automatic formatting of scripts. If you've never programmed before, you might not understand that indentation is a way of indicating the hierarchy or order of processing of various sections of your scripts. Experienced programmers learn to set their code so that it forms neat and symmetrical lines that are logical and easy to follow on the page.

The script editor forces your text into the proper indented form for HyperTalk's syntax. Auto-indentation happens when you close the script by pressing the OK button. You can force auto-indentation to occur by pressing the Tab key. In the case of our `on mouseUp, beep, end mouseUp` script, the beep was indented. Indenting is not just an aesthetic consideration. Certain syntax errors throw off the indenting, resulting in a handler that is flush left at the top but not at the bottom. This is an indication that something is amiss in your script.

Message Disagreement

This is a simple error but you will be amazed at how often it can crop up. If you have a different message name after the `end` keyword than after the `on` keyword, indentation won't line up properly. Delete some characters from either occurrence of `mouseUp` in the New Button script and press the Tab key, and you see what we mean. Pressing the Tab key makes the editor reformat the script. When we get into more complicated scripts, you will find that it is a good idea to press the Tab key now and then to see if things are in order.

Option-Return for Long Lines

The width of the script editor is limited to about 65 characters. If the line of the script reaches the right edge of the script window, typing continues to move the cursor, and characters are entered even though they won't be seen. The end of a statement in HyperTalk is always signaled by a return character at the end of the

line. How can you type a long HyperTalk statement and still see it? The answer is the option-return character. When you reach the right side of the editor window, and you wish to continue the statement on the next line, simply press option-return. A ¬ special character will appear at the end of the line, meaning "this line is continued on the next line." Many of our scripts use ¬ to make them more readable. Get used to using it—your scripts will be easier to read and use.

The HyperCard-HyperTalk Relationship

HyperCard and HyperTalk are wonderfully integrated, as you can see in the next example.

Example 5:

1. Open your HyperTalk Bible stack.
2. Choose the button tool and open the New Button you created in the first example. With the button selected, double-click to open its info dialog box, and change its name to `pipe`.

   Remember, the button's script contains the simple `beep` we entered in the previous example. By the way, from now on we won't tell you to switch to the button tool to open a button's script or the field tool to open a field script. We assume you will do that automatically, either by using the menu to change the tool or by using the command-option keys.
3. With the button dialog box open, click the LinkTo... button. A small windoid called Go to destination, then aim link at: with three buttons in it appears.
4. Go to the home card by typing Command-H, then click the Art Stack. Your screen looks like Figure 1-6, with the windoid following you around. Press the This Card button.

Result:

You are returned to the original card that initiated the LinkTo... step.

5. Select the browse tool, and click the pipe button. You are taken to the card in the Art Stack with the man smoking the pipe.
6. Go back to the HyperTalk Bible stack card (Command-~ does it), and open the script for the pipe button.

Result:

The pipe button should be changed to

```
on mouseUp
  beep
  go to card id 2996 of stack "Clip Art"
end mouseUp
```

HyperCard automatically added some script to our original mouseUp handler! The statement `go to card id 2996 of stack "Clip Art"` is HyperTalk for what it sounds like: go to the Clip Art stack, and then to `card id 2996`. (Each card has a unique identification number (ID) written as `id`). The HyperTalk command `go to` is one you learn more about in chapter 2, "Objects," but our point here is how tightly HyperCard's environment is integrated with its internal language ability.

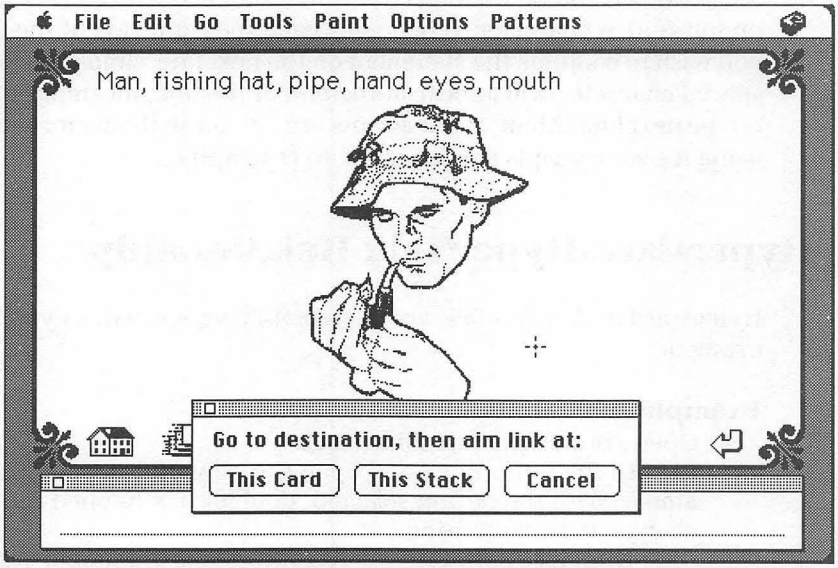


Figure 1-6. Linking to the pipe button.

Where's the Script?

Buttons are the most natural place to put scripts, because the whole idea of buttons is that they make things happen without limiting you just to placing scripts in button objects. Every HyperCard object has a script property and can become a repository for HyperTalk instructions.

To demonstrate the use of scripts in other objects, place a very simple handler into the script of the Getting Started card in our HyperTalk Bible stack. A card script is just like a button script, i.e., it exists inside an object and responds to external messages.

We also introduce another HyperTalk command, `play`, so you can start making sounds that are more comfortable than the jolting beep.

Example 6:

1. Open your HyperTalk Bible stack.
2. Open the card info dialog box and click the Script... button.
3. Enter the following handler in the script editor:

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

4. Close the editor and click anywhere on the screen outside the button called pipe.

Result:

Each time you click the mouse button, you hear a boing sound.

In this example, you created a handler from scratch without the assistance of the handler outline provided for buttons. With the `play` command, you can make interesting sounds. We're not even through the first chapter and already you are making sounds that are beyond most of the programs on the market. You learn a good deal more about the `play` command in chapter 3, "Audio and Visual Effects," but if you want to play around with it now, there are two `play` voices included with HyperCard: boing and harpsichord. You can use either one for interesting effects.

Getting the Message

Appearances to the contrary, there are many more Hypertalk messages than just `mouseUp`. We have already mentioned `mouseDown` in passing and there are other mouse-related messages, a number of object-related messages, and some messages that come directly from HyperCard itself. We won't get into these messages yet, but are saving them for a more exhaustive discussion in chapter 6, "Messages." Suffice to say, there are messages for more elaborate tracking of the mouse, for letting you know if a card is open or closed, and for the deletion of any object. There are `startup` messages and `quit` and `idle` messages.

Making Comments

Often, you want to make notes about how your scripts work, when you altered them, and so on. You want these notes in the script where you, or someone else, can see them when they look at the script. Notes in a HyperTalk script are called *comments*. They don't do anything except display the text in them. Indicate that HyperCard should ignore the text by putting two hyphens in front of it. Everything, whether a few words or an entire line, to the right of the comment hyphens is ignored. A commented handler might look something like this:

```
-- Tone button, © December 31, 1988, The Waite Group, Inc.

on mouseLeave            -- handler runs when cursor leaves button
  play "harpsichord"     -- makes a musical tone
end mouseLeave
```

There is another use for comments in HyperTalk. You can disable a line in a script temporarily by putting comment hyphens in front of it. This comes in handy when you *debug* a script. Debugging means finding what is wrong with your script. The word comes from the old days when computers used vacuum tubes instead of transistors and integrated circuits. Apparently, a moth flew into the computer, was fried by the heat, and caused the computer to break down. The person fixing the computer found the bug and since then the word "bug" has been used for all forms of errors that occur in a program.

Here is a handler with a commented play command temporarily replaced by a beep command:

```
-- Tone button, © December 31, 1988, The Waite Group, Inc.

on mouseLeave            -- handler runs when cursor leaves button
  -- play "harpsichord" -- this is disabled
  beep                   -- let's try this for a moment
end mouseLeave
```

The Message Box—Your Scripting Control Panel

Before we get too enveloped in HyperTalk scripting, you should learn more about the Message box, a simple and nifty tool that makes learning scripting much easier. Remember, the Message box is really a small windoid into which you can type HyperTalk commands and statements that you wish to run immediately. (See Figure 1-7.)

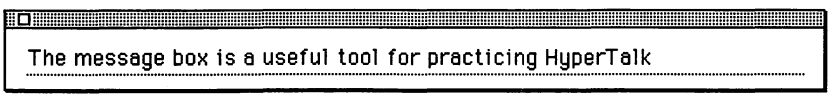


Figure 1-7. The HyperCard Message box.

Not only can you type commands into the Message box, but you can also use it as a complete calculator. As you will see, the Message box can also be a place to display items from your scripts. In a way, the Message box is more like a display and control panel for scripting and output from scripts.

Normally, HyperCard hides the Message box when it starts up. (The box might have been open the last time the stack was used, so you'll often see scripts that start out by hiding it just in case.) Bring the Message box into view any time by pressing Command-M or by choosing Message from the Go menu. Close the Message box by typing Command-M again or by clicking the Message box's tiny close box in the upper-left corner.

Example 7:

1. Open the HyperTalk Bible stack.
2. When the card appears, type Command-M to reveal the Message box.
3. Type beep 3 into the Message box.
4. Press the Return key.

Result:

You hear three rapid beep sounds in succession, producing a machine gun effect.

That is all there is to using the Message box. You can experiment with changing the number after the `beep` command. Because it is such a versatile and useful feature for practicing scripting, HyperTalk commands, and even navigating, we rely a great deal on the Message box in this book.

Message Box Tips

The Message box is a window that can be moved anywhere on the screen, but can't be resized. In scripting, you can program the location of the Message box to be anywhere on the screen, and even off the screen entirely. There are several important characteristics of the Message box you should know about as you begin your HyperTalk scripting.

If there is text already in the Message box, and the blinking text cursor is not showing in the box, the next thing you type on the keyboard will go directly into the box. As soon as you type your first character, the text insertion cursor appears.

If you enter a command in the Message box, you can repeatedly press the Return or Enter key to execute that command over and over. This feature lets you try repeating long commands without having to retype them. If some word in the line in the box is typed wrong, simply click the word, correct it, and press Return or Enter again. Clear out the contents of the Message box with the Clear key found on the numeric pad on the Macintosh Plus, Macintosh SE and Macintosh II.

One slightly obscure feature of the Message box is that it doesn't need to be visible to be used. You can immediately begin typing, and, if you don't have the cursor in a text field and are in the browse mode, anything you type will go into the hidden Message box. So if you type `beep` and press Return, HyperCard will beep at you. To use this feature, you must be in the scripting mode and the Blind Typing box in the Home stack's User Preferences card must be checked. Blind Typing is useful when you don't want the Message box to obscure something on the screen but you still want to use it.

In scripting you can abbreviate the name Message box, as `msg box`, or just `msg`.

The HyperTalk Vocabulary

We discussed the syntax or vocabulary of HyperTalk in the Introduction to this book. The complete vocabulary is quite large, consisting of about 100 special keywords. Throughout this book, you will learn this vocabulary, and, as in learning any foreign language, the better you understand the vocabulary, the better you speak the language. Thus, the better you understand the vocabulary of HyperTalk, the better your scripts are. We already learned one vocabulary word: the `beep` command. Let's learn another more useful word.

Vocabulary Expansion: *put*

One of the most frequent things you do in any language, be it HyperTalk, English, Fortran, or Froginese, is present or display information—text, graphics, letters, or apples. You may want to fill some fields in a card with numbers, names, fortunes, or gas levels in an experiment, or you may wish to present volumes of text in a friendlier format (using graphics and other fancy effects) to make the material less cold and more exciting to read. HyperTalk has the `put` command for accomplishing such things.

The Syntax

The `put` command has a certain syntax or format that must be followed for it to work properly. In this book, we illustrate these formats for all the keywords in a concise description of all the options that can be used with the commands. The HyperTalk syntax for `put` is illustrated in Figure 1-8.

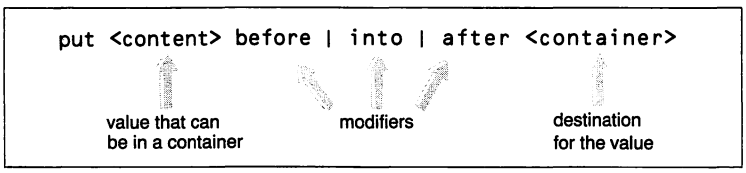


Figure 1-8. The `put` command syntax.

Items that must be supplied to the `put` command for it to work are surrounded by greater-than (>) and less-than (<) symbols. In the `put` command syntax, the item `<content>` comes after the word `put` and is described as being a value or a *container*. This means that you can type the number `5` after `put`, or you can type a word surrounded by quotes, such as `"pizza"`. Although you have not been exposed to this concept yet, a container is an object that can hold (contain) text or numbers. It can be a field, the Message box, or, as we shall see later, a variable. Fields and the Message box are the most obvious containers, because they are visible objects. For example, you can follow `put` with the name of a field.

Three *modifiers* follow the put command: `before`, `into`, and `after`. These are separated by the bar character (|) to indicate that only one of them can be used at a time (the bar means or). Following the modifiers is the *destination* for the `put` command, called `<container>` in the syntax to indicate that this must be a legal container for `put` to work. This is where the `<content>` will go. The modifiers allow you to place the value in `<content>` either before, into, or after what is in another container. If you have a statement (sentence) in HyperTalk like

```
put 5 into msg box
```

the value `5` is placed into the Message box and any previous text in the Message box is erased and lost.

But if we say

```
put 5 after msg box
```

the number 5 is placed immediately after any text already in the Message box. If we type

```
put 5 before msg box
```

the number 5 is placed before any previous text in the Message box. Later, when we're writing scripts, these modifiers are extremely useful.

By the way, HyperTalk is very forgiving about leaving things out. For example, if you leave `msg box` off the end of the `put`, it just assumes that is where you want the `put` to go. So,

```
put 5
```

simply puts the value 5 in the Message box.

Using *put* and the Message Box

Let's see how to use `put` in a simple experiment using the Message box.

Example 8:

1. Open your HyperTalk Bible stack.
2. The first card shows the button named pipe.
3. Move the button over to the left as shown in Figure 1-9.
4. Make a new field by selecting New Field in the Objects menu. The new field appears with marching ants around it. Be careful not to click outside the field as this deselects the field and, because it defaults to a transparent style, it is impossible to see.
5. Double-click the new field to open its dialog box, name the field `myField`, and set its style to shadow. Position the field as shown in Figure 1-9.
6. Type Command-M to open the Message box if it is not already showing. Enter

   ```
   put "roger wabbit" into card field "myField"
   ```

   in the Message box and press the Return key.

Result:

The words `roger wabbit` appear in the card field.

When you enter a legal HyperTalk statement, or command, into the Message box, it attempts to run that command as if it were inside a script. Here, the statement `put "roger wabbit" into card field "myField"` placed the string of characters `roger wabbit` into the card field we just made, myField.

The `put` command used the modifier `into` so that the old contents of the destination container are completely replaced with the new. Let's try another option.

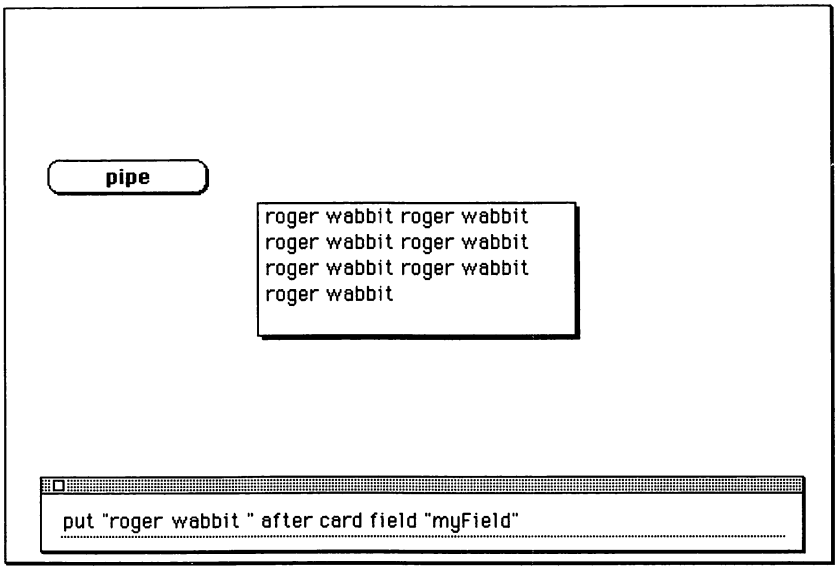


Figure 1-9. Using `put` to enter words into a field.

7. Change the word `into` to `after` and add `& space` so that it reads `put "roger wabbit" & space after card field "myField"`, and press the Return key several times.

Result:

The words `roger wabbit` appear in the field each time you press the key.

Use *into* to Erase and Replace

When you use the modifier `into` with `put`, you erase any information already in a container. Use the modifiers `after` or `before` when you wish to append information to the container. Use the `space`, `return`, and `&` constants to properly separate the items you put `before` or `after`. An example is

```
put return & "This is the new text" & return after myfield
```

which forces a carriage return to be placed before your new text so that it's separated from the previous line.

In this example, we replaced the optional word `into` with `after`, which tells `put` to place the text after any text already in the field. This is a nondestructive write to the container. We also added the code `& space` after the text `roger wabbit`. This causes a `space` character to be added to the end of roger's name. The `&` does the attaching for us and guarantees that all words are separated by a space. `space` is a HyperTalk constant. You learn about most of these constants when we get to math in chapter 5. Experiment with changing the keyword to `before`.

More *put* and a Trick

Let's see what we can do with `put` in a button.

Example 9:

1. On the Getting Started card, create a new button, set its name to `Rabbit`, position as in Figure 1-10, and enter the following script in it:

```
on mouseStillDown
  put "≡ ≡ ≡ ≡" into card field "myField"
  put " ≡ ≡ ≡ ≡" into card field "myField"
end mouseStillDown
```

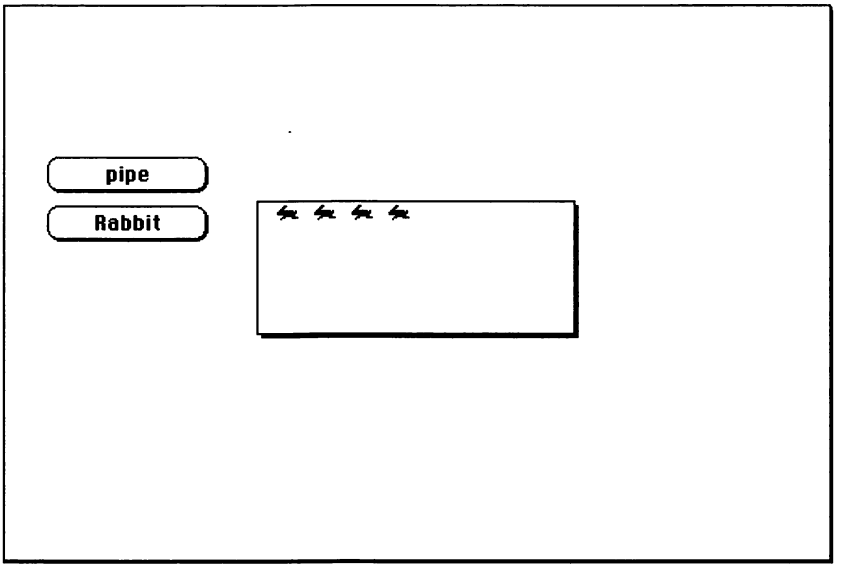


Figure 1-10. Jumping rabbits.

The characters that look like ≡ can be obtained by pressing the Shift, Option, and tilde (~) keys on the Macintosh keyboard all at once. HyperTalk's script editor shows a hatchmark on the Macintosh screen for these keys. However, the actual output depends on the font of the field the character is placed in. In the Monaco font, it makes a candle; in Geneva, it makes a rabbit; in New York, it makes a robot, etc. Note there are two spaces after each ≡ character, and there are two spaces before the first ≡ character in the second `put` statement.

2. Return to the browse mode and hold down the button on the mouse.

Result:

A rabbit appears to jump in the field.

The `mouseStillDown` message works differently than `mouseUp`. As long as the

mouse button is held down, the message calls the handler. You can repeat a statement or group of statements over and over as long as the mouse button is held down. In our example, the ≡ character, when sent to the General default font, causes a tiny rabbit to appear in the field. The two `put` statements have the character strings offset, so it appears that the rabbit is moving to the left.

Switching between Tools

HyperCard Version 1.2 allows you to switch quickly between the browse, button, and field tools by using keyboard shortcuts. Command-Tab selects the browse tool, Command-Tab-Tab selects the button tool, and Command-Tab-Tab-Tab selects the field tool. Using these keys instead of the tool palette saves you time.

Vocabulary Expansion: *set*

We mentioned in the Introduction that the concept of properties is one of the cornerstones of HyperCard. We have seen that `put` is used for putting information into fields or the Message box. What about controlling properties? Do we use `put` to change them? No. HyperTalk has a separate command for controlling properties called `set`. The different command makes it less likely you will misuse the syntax. The `set` command syntax is illustrated in Figure 1-11.

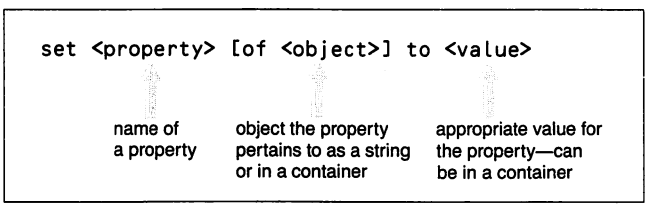


Figure 1-11. The `set` command syntax.

The syntax for `set` shows that it comes first in a HyperTalk statement, as does `put`. It is followed by the name of the property we wish to affect, here called `<property>`. There are dozens of properties for each object in HyperCard and part of your mastery will come from understanding all of them. Some obvious property examples are the font and style of fields and buttons, and some less obvious ones are the name and icon of a button.

The name of the property is followed by the words `[of <object>]`. The brackets tell you that the words between them are optional and don't always need to be included for the command to work. This means you can type

```
set <property> to <value>
```

or you can type

```
set <property> of field 1 to <value>
```

The final item `<value>` holds the value you wish to set the property to, the name of a font (like Chicago), for instance, if you are changing the font property of a field, or the style name of a button (like transparent), if you are changing the style property of a button. As the syntax shows, the `<value>` can be a container or a fixed value.

Example 10:

1. Create a new button, set its name to `Jumping`, position as in Figure 1-7, and enter the following script in it. (Note the use of the option-return character because the line is so wide.)

```
on mouseStillDown
  set the loc of card field "myField" to ¬
random(100)+200,random(75)+100
end mouseStillDown
```

2. Return to the browse tool and remove all but one of the rabbits from the field myField.
3. Switch to the field, too. Open the info dialog box for the card field "myField" and change its style to transparent. With the info box still open, change the font size to 24-point, and resize the field so it just surrounds the rabbit. See Figure 1-12.

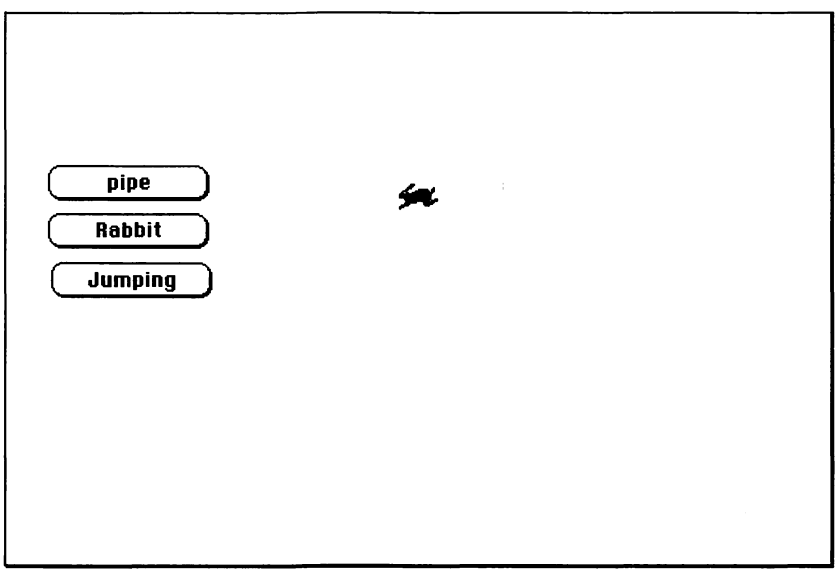


Figure 1-12. Using `set` to control the rabbit.

4. Return to the browse tool and click the Jumping button.

Result:

The rabbit jumps all over the place as long as you hold the mouse button down.

With this example, we change the location property of the card field inside the mouseStillDown handler. That is the purpose of the statement `set the loc...`. The location `loc` is computed by a special HyperTalk function called `random`. It is used for generating random numbers, and here it chooses the x and y locations for the center of the card field "myField".

Card Copying Is Dangerous

One of the weirdest things about HyperCard is that when you try to copy a card from one stack to another, you always end up copying the background of the original card with the card itself. When you paste that new card into a different stack, the new background comes with it. This wreaks havoc on your stack's organization and even makes it inoperable. The only way around this is to copy each of the objects on the source card to the destination card one at a time. The graphics can easily be copied in one step, but if there are many buttons and fields, this copying is laborious. The public domain script called Groupies written by Sioux Lacy with improvements by Bruce Barrett, is used to move groups of buttons and fields using HyperTalk. It is available on AppleLink.

As the syntax for `set` shows, the reserved words `of object` are optional because in some cases we don't need to say what the property applies to. There are many properties we will be discovering and in all cases the `set` command will be used to control these properties.

Summary

In this chapter, you learned the basic concepts of the HyperTalk script editor and created your first script with it. You learned the rules of upper- and lowercase, how to use the Message box for quickly trying out single HyperTalk commands, and two new vocabulary words, `put` and `set`. We also showed you how to set up a practice stack, what version of HyperTalk to use, and how our syntaxes work. You are now in a good position to take the next step in HyperTalk scripting: learning to use objects.

Review Questions

All chapters in this book, except the Introduction, contain a set of review questions. Answers are provided so that you can check your understanding of the topic.

1. Which of the following best describes the approach taken by this book?

a. presents HyperTalk in a linear manner for programmers familiar with languages like Ada or Forth, and provides details on building a word processor in HyperTalk
b. presents HyperTalk for people who have never programmed before or who have some experience with a language like BASIC, Pascal, or C, and uses a spiral approach
c. presents advanced topics in HyperTalk, including tricks of the master scriptors

2. Which versions of HyperCard are compatible with this book?

   a. 1.1
   b. 1.2
   c. 1.2.1
   d. 2.0
   e. 1.2.2

3. What user level should HyperCard be set to for learning to script?

   a. authoring
   b. painting
   c. browsing
   d. scripting
   e. finding

4. Which are the best settings of the Power Keys and Blind Typing preferences?

   a. both off—they are wasteful of memory
   b. both on—they add two neat features to HyperCard
   c. Blind Typing on, Power Keys off—you can be blindfolded and still type into HyperCard
   d. doesn't matter—these are options which don't affect your scripting

5. Which of the following best describes the way to organize your stacks for this book?

   a. Put your HyperTalk Bible stack into the same folder as the Home stack.
   b. Put your HyperTalk Bible stack into the same folder as the Home stack, and if you have a hard disk, put all the HyperCard folders and the application HyperCard into a folder called HyperCard.
   c. Put your HyperTalk Bible stack into a folder called HyperTalk Bible and put this folder at the same level as HyperCard.

6. True or false? Every object in HyperCard has a property called the script.
7. True or false? A script editor is a powerful word processor you can use for writing large documents.
8. What does the following handler do?

```
on mouseUP
 beep
 beep
 beep
end mouseUp
```

9. All handlers begin with the words `on something` and end with `end something`. If you leave the `end something` off, the handler will

   a. work just as if you included it
   b. beep the speaker and give an error Message box
   c. run in a loop and cause you to restart the Macintosh
   d. permanently damage your Macintosh

10. True or false? Upper- and lowercase are ignored by HyperTalk.
11. If a script refuses to indent properly in the script editor, it means

    a. you have a statement or word that HyperTalk does not understand, or you have left off an `on something` or an `end something`
    b. your version of HyperCard is incorrect
    c. there is not enough memory in the computer

12. If a HyperTalk line in the script editor is too wide to be visible, you can fix it by

    a. typing Command-return at the end of the line
    b. typing option-L at the end of the line
    c. typing option-return at the end of the line

13. True or false? HyperTalk handlers can only be in button or field objects.
14. The way to indicate a comment in a HyperTalk script is with

    a. `**`
    b. `|`
    c. `/*`
    d. REM
    e. --

15. True or false? The Message box is not very useful because it does not allow you to type in HyperTalk commands.
16. You can open the script of a button with command-option click, and a field with shift-command-option click. Can you close the editor with command-option-click?
17. Which statement best describes HyperTalk's vocabulary:

    a. HyperTalk uses a vocabulary much like Spanish, and has constants like uno, dos, tres, and so on.
    b. HyperTalk is much like Esperanto, a universal vocabulary anyone in the world can understand, read, and write.
    c. HyperTalk is a computer language very much like the English language. It has verbs and objects, adjectives and prepositions.

18. Which of the following is an illegal use of the `put` command?

    a. `put "loud mouth"`
    b. `put 1 + 2 into msg box`
    c. `put beep into msg`
    d. `put into msg`

19. The words `before`, `into`, and `after` in the syntax for `put` are called

    a. adjectives
    b. modifiers

c. ancillaries
d. position holders

20. What does the following handler do?

```
on mouseUp
   play "boing"
   put "...heres the Message box"
end mouseUp
```

21. What does the key combination shift-option-tilde do?

    a. It provides a special graphics character in each Macintosh font that can be used in your scripts.
    b. It reboots the Macintosh.
    c. It adds a debugging mode to HyperTalk.
    d. It allows windows to be resized.

22. True or false? `mouseStillNotDown` is a valid mouse message.
23. Which of the following statements best describes the set command's purpose?

    a. The `set` command is used to set HyperCard to work on the Mac II.
    b. The `set` command controls the size of the set of colors available on the Mac II.
    c. The `set` command is used to change properties of objects.

24. When typed into the Message box what does the following handler do?

```
set the loc of the Message box to 0,0
```

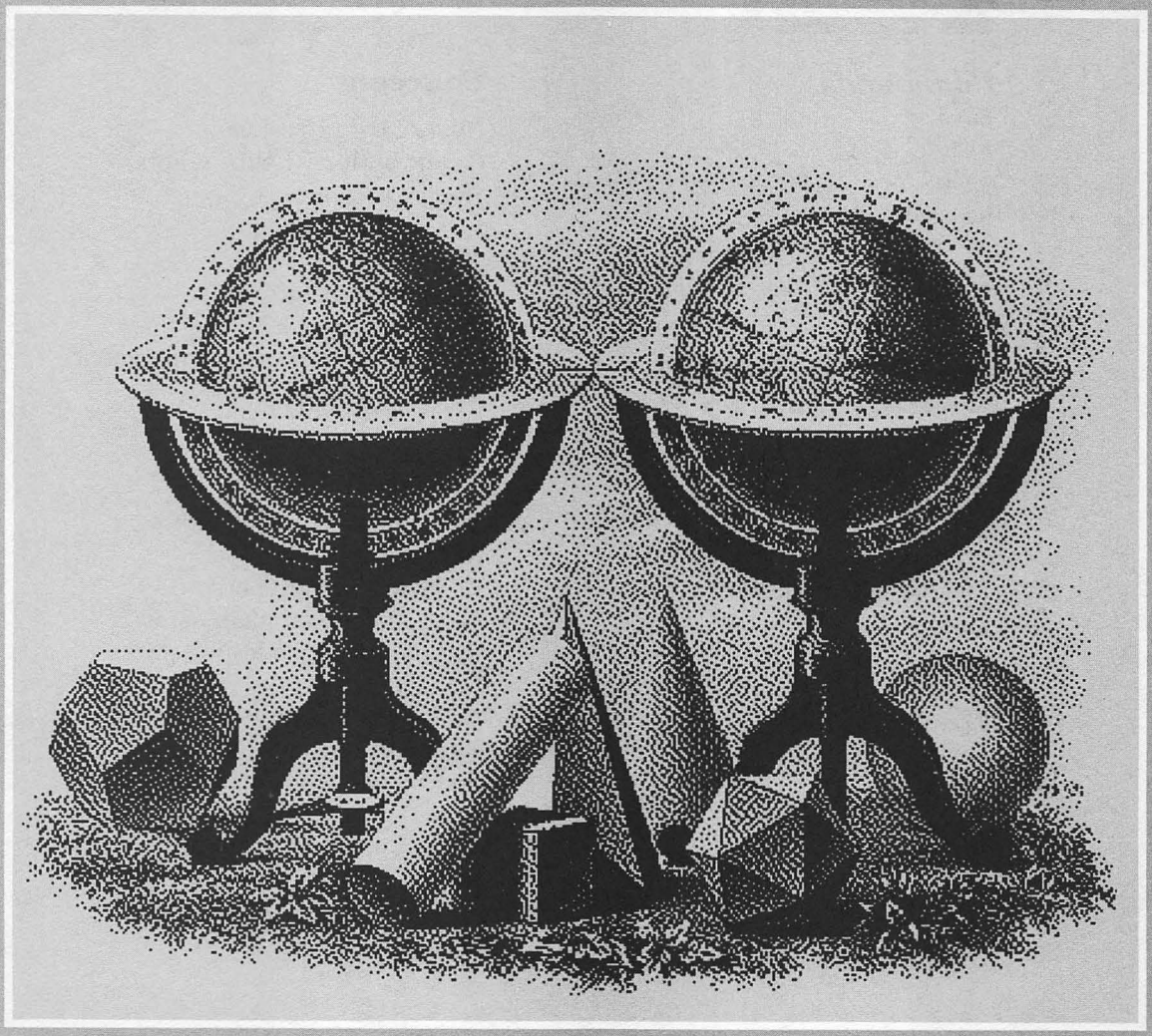
Exercise

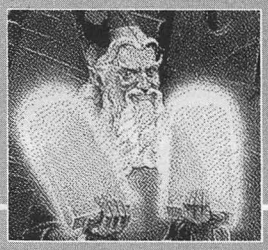
All chapters have exercises to test how well you have learned the contents of the chapter. These start simple and grow more elaborate as the chapters progress and more vocabulary is understood.

Write a script in a button that beeps the speaker and moves the button to a random location on the screen.

Chapter 2

Objects





Quick View

Commands

pass
hide
show
go card
go stack
push
pop

Objects and Modifiers

me
this

Functions

target

Ordinals

any
mid
last
first
next
prev

Concepts

HyperCard refresher
Cards, buttons, fields, graphics
HyperCard's layers
Using card elements
The stack
Object hierarchy
Passing messages
The galactic HyperCard metaphor
Naming objects
Names
Relative numbers
ID numbers
Showing and hiding fields
Improving pop-ups
Using me, target, and this
Long, short, and plain names
Pathnames: zipcodes for stacks
Navigation
Navigating within HFS
push and pop
Special descriptors for navigation

Chapter 2

Objects

Mastery of HyperTalk is intimately tied to understanding its built-in objects. These objects—fields, buttons, cards, backgrounds, and stacks—have features that a HyperTalk programmer must know to manipulate them properly and take full advantage of their capabilities. You need to understand how a stack is organized in HyperCard, how cards work, and how the objects within that stack are identified and accessed from the language.

For example, you can't do a good job of getting HyperTalk to move you from one card to another unless you understand how cards are identified as objects. Likewise, to locate button and field objects on other stacks, you need to create a *path* to them, that is, find which card they exist on and even what stack they may be in.

In this chapter, we explore the fundamental concepts behind objects, and we also fill you in on more of the fundamentals of HyperTalk. Although this won't be an exhaustive discussion because you are still in the "spiraling" mode, you learn how messages flow between objects (the hierarchy), how objects are named and identified, how to access them in different stacks (pathnames), and how to navigate between card objects. You also learn how to do a little object hide and seek and make fields and buttons disappear. By the time this chapter is over, you will be well on your way to writing your own stacks.

HyperCard Object Refresher

Most of this material will be familiar to the HyperCard user who has been authoring, making stacks without scripts, but it is important to make sure that, as a scriptor, you understand these basic concepts.

Cards

HyperCard is based on the idea of single *cards* that fit snugly in a Macintosh Plus or SE 9-inch screen, as shown in Figure 2-1. When you first look at the card, it appears it is

not the normal Macintosh window that applications usually present. A normal Macintosh window has a title bar that tells you the name of the opened document. Where is this in the HyperCard card? Actually, Mac developer Bill Atkinson performed a little trick here. There is a title bar—it's just hidden from view on the SE and MacPlus screens. The title bar is located above the card, in an area that doesn't show on the smaller 9-inch screens. If you move the card down, it will come into view. People with large screen Macs see this immediately when using HyperCard. Example 1 shows the HyperCard title bar. (Large-screen users don't need to do this example. It moves the card up to the extreme upper-left corner of your large display.)

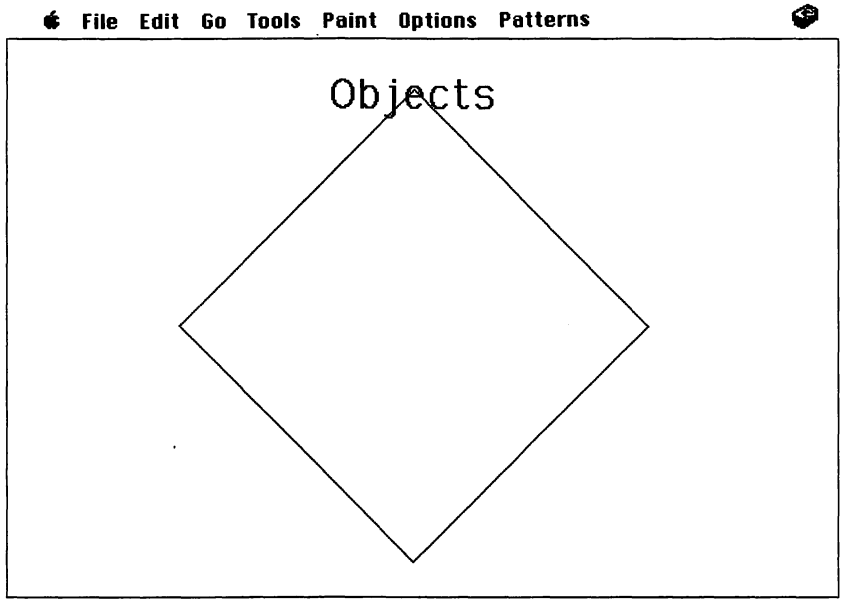


Figure 2-1. A HyperCard card as seen on an SE or MacPlus.

Example 1:

1. Open your HyperTalk Bible stack. Create a new card after the Getting Started card, by selecting New Card from the Objects menu, or pressing Command-N.
2. Name the new card `Objects`.
3. Select the paint tools by clicking the polygon tool in the Tools menu. Put the cursor in the center of the card and draw a diamond shape like that shown in Figure 2-1.
4. Select the text tool and type the name `Objects` on the card as shown in the figure. Change the font to 18-point Geneva bold.
5. Bring out the Message box with Command-M, enter the following statement, and press the Return key:

```
set the loc of card window to 0,40
```

Result:

The card moves down 40 units vertically, exposing its title bar (see Figure 2-2).

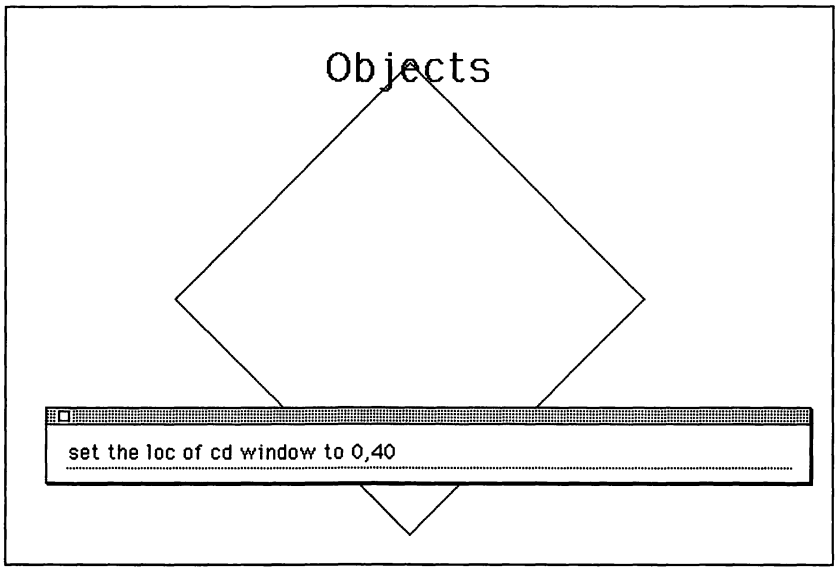


Figure 2-2. A HyperCard title bar.

6. Change the 40 back to 0, so the Message box contains

```
set the loc of card window to 0,0
```

Result:

The card moves back to its old location.

You can also turn the menu bar on and off to increase the area visible on the card. (Command-space does this.)

Think of cards as *screen objects*, somewhat like button and field objects. They act like a collection of index cards that can store graphics information. Text can be "painted" onto cards with the paint tools. You can easily move back and forth between cards. Cards can't be resized like normal Macintosh windows, however. That is the constraint that the creator of HyperCard set to distinguish HyperCard from most applications. (It is possible newer versions will have resizable windows.)

HyperTalk cards can contain three basic building elements: buttons, fields, or graphics (see Figure 2-3). All three can be resized, moved, copied, and made invisible. Buttons come in seven styles, fields in five styles.

Cards

Theoretically, HyperCard allows up to 16,777,216 total empty cards in a single stack! However, because the objects, bit-maps, and text on a card contribute to this limit, as these items grow, the number of possible cards diminishes. Because HyperCard loads an entire card into memory, the actual limitations are set by the number of bytes of available memory. It is

possible that a card with lots of text, objects, and bit-maps would load and run on a 2M SE and not run or fit in a 1M Mac Plus. HyperCard also allows the card script to be up to 30,000 characters and the name of the card to have up to 31 characters. Because all these different objects contribute to the size of a card, the current maximum useful size of a background or card is between 50 and 100 kilobytes.

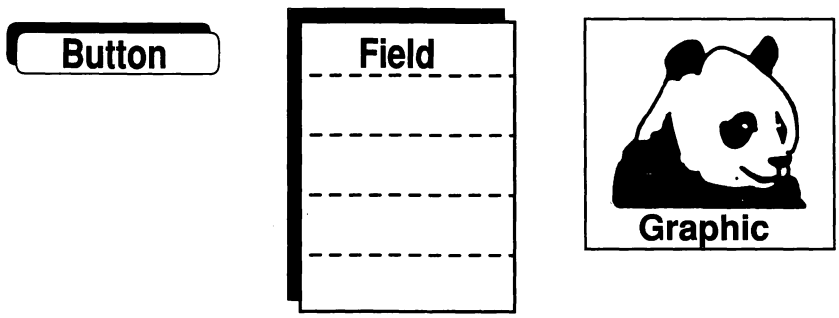


Figure 2-3. The three basic elements of HyperTalk cards.

Buttons

Buttons, which, as you know, can contain HyperTalk instructions, can be set from the dialog box to one of seven different styles: transparent, opaque, rectangular, shadowed, round rect, check box, or radio. The rounded rectangle button is the most common and is the default when you choose New Button. It is the one with the double rectangle around it. The OK button in Figure 2-4 is an example of a rounded rectangle.

Buttons can be made invisible, given a unique identifying name, and told whether to be "hilighted" or not. They can be linked to other cards, given a unique icon from a built-in list of HyperCard tools (see Figure 2-5), moved and positioned, and can contain a script. You can control whether the name of a button shows up in the button on the screen. Much of what you learn in HyperTalk scripting will end up inside a button.

Buttons
Theoretically, HyperCard allows up to 32,767 buttons per card or background. The largest script in a button can be 30,000 characters. Note that it is probably very impractical to use such a large number of buttons. Indeed, at approximately 100 buttons on a card, HyperCard becomes very slow, even on the fastest Macintosh.

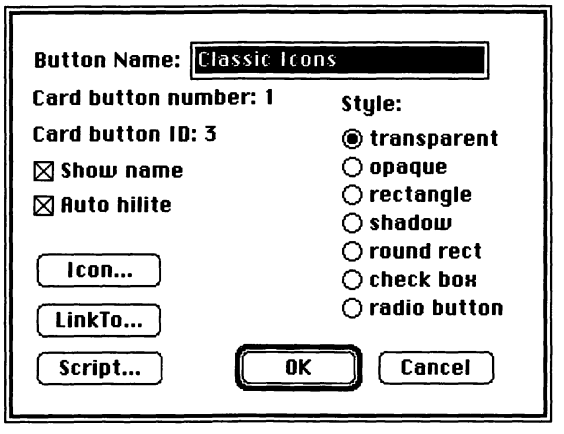


Figure 2-4. The button dialog box.

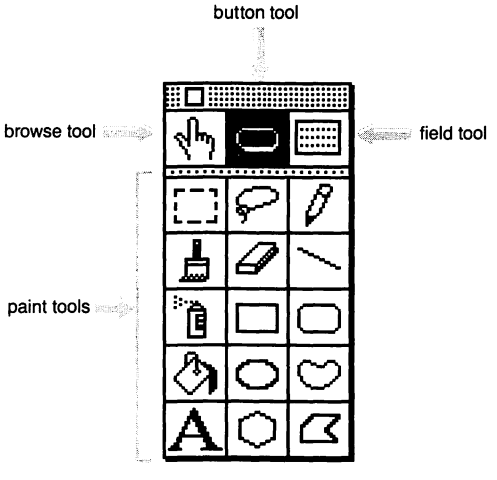


Figure 2-5. The HyperCard tools.

Fields

Fields come in five different styles: rectangular, opaque, scrolling, shadowed, and transparent. See Figure 2-6. They can be named and can be positioned and resized, as well as given a script, like buttons.

Fields don't link to other cards, or "autohilite," but they have check boxes that control the lock text property. This means that you can set a field to not allow anyone to type into it. You see later how this allows us to put handlers into fields and have them act like buttons. There are check boxes to control whether or not you can see lines in the field, how wide the margins are, and whether autotabbing to this

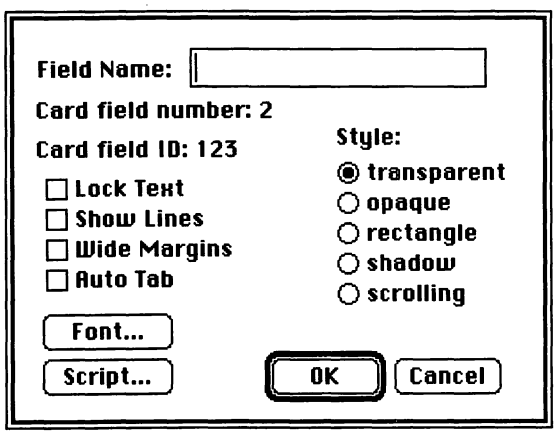


Figure 2-6. The field dialog box.

field from another field is to be allowed. You can also set the font in a field to any font in your system folder.

Fields
Theoretically, HyperCard allows up to 32,767 fields per card or background. The largest script in a field is the same as a button, up to 30,000 characters. The largest amount of text is 30,000 characters. (Don't get pessimistic: regular HyperTalk variables can be huge, limited only by memory available. More about that when you learn about variables.)

Graphics

Graphics can be applied to a HyperCard card as if it were a small MacPaint document. You can place transparent (see-through) or opaque (not see-through white or black) pixels on a card or on a background.

Graphics
Theoretically, HyperTalk allows up to 175,104 pixels to be painted on each card. This is derived from the fact that the dimensions of a card are 512 × 342. When you consider that HyperCard packs all these pixels onto the hard disk into 30 times less space, you can appreciate what a powerful tool this is. Appendix E of the *HyperCard Script Language Guide** claims the maximum size of a bit-map is 44 kilobytes; 175,104 pixels consumes 21,888 bytes, or about 21 kilobytes of memory. The extra 22K bytes is used to hold the transparency/opaque state of each pixel.

*Written by the Apple Computer Inc. staff, this guide was published by Addison-Wesley Publishing Company in 1988.

HyperCard's Object Layers

Each object in HyperCard has its own layer. If it is transparent, you can see through the layers beneath it. If the object is opaque, it blocks you from seeing any other objects directly beneath it. The best way to visualize this layering effect is to think of each object in HyperCard as having its own clear plastic sheet the full size of a card. Each time you add another object, like a button or field, you add a new sheet of plastic to the stack (with your object on that layer). Any opaque object will be seen through the stack if there is not another opaque object in a different layer in front of it. If you add a transparent object, the objects underneath it in other layers show through. (Using the command-option keys allows you to see any transparent buttons or fields.)

If an opaque button is beneath a transparent button, the opaque button shows through the transparent button, but the transparent button receives the button click of the mouse. You can move the location of an object's layer with the Command-equals and Command-dash keys, which cause the object to move forward or backward, respectively.

There are two basic kinds of stacks: homogeneous and heterogeneous. A *homogeneous stack* has more than one card but only one background. Every stack has at least one card and one background.

Background and Card Layers

Not only does HyperCard support layers for objects, there are two additional domains for holding these objects: the *background* and *foreground*. You can see the background for a card at any time by pressing Command-B. The menu bar is outlined with hatchmarks, and only the parts on the background card are visible. Pressing Command-B a second time takes you back to the foreground layer.

Backgrounds are unique in that anything you place on the background (button, field, or graphic) is visible and active on every card that is in a homogeneous stack. Objects assigned to the background are referred to as background objects, and objects on a card are called card objects. When objects are assigned to a card, they sit on top of all the background objects, so you can think of card objects as foreground objects. An opaque button on a card is visible and active on the card it is on. See Figure 2-7.

Graphics can be on cards or backgrounds. In Figure 2-7, the panda is a card graphic and the frame is a background graphic. Background buttons show on all cards, but card buttons do not show on backgrounds.

A unique and nonintuitive feature of background fields is that the text that is placed in them is not the same on every card, but rather is unique to each card. Any text you enter into a background field is visible only on that particular card. The properties of the field don't change, of course, so the font, style, and other properties stay the same from card to card for a background field. Backgrounds do not share objects as cards on a background do, so if you want identical objects on different backgrounds, you have to make separate objects for each background.

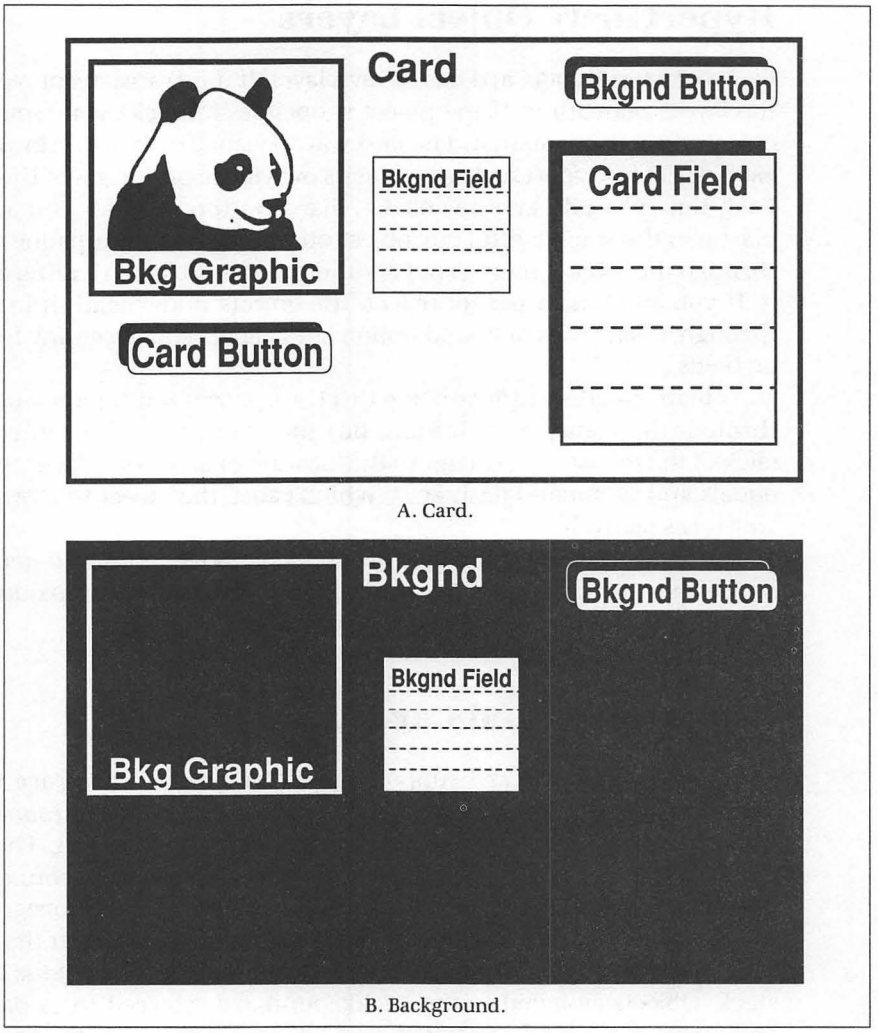


Figure 2-7. Cards and backgrounds.

One of the best ways to control what a browser of your stack sees is to use opaque objects like buttons and fields to hide background objects. Graphics can also be used to block underlying objects. In Version 1.2 of HyperCard, four new commands for showing and hiding a card or background layer were added. These commands give a fair degree of control over what the browser of your stack sees. However, HyperCard is not strong on graphics support. Although you can have 32,767 individually controlled parts on a single card that can be shown or hidden from view, you can only have one graphic layer.

A *heterogeneous* stack is a stack with more then one background. Each card is attached to a unique background. Cards in such a stack continue to be arranged sequentially, but now each card has a distinct background that shows through it.

There are many ways to arrange a heterogeneous stack. It could be set up so that when you move from one card to another the background stays the same for the first 10 cards, then changes for the next 10, and so on. Or, you can set up a stack so that the background changes every other card, or however you want. The background a card is in depends on what background was showing when the card was first made. A card inherits a unique background.

Using Elements of a Card

Usually when you build a stack, you choose elements that are common to appear in the background and elements that change to be assigned to cards. In Figure 2-7, our animal stack has the frame for the animals on the background and the graphic for an individual animal on the card itself. Each animal is on a distinct layer. Although each card obscures the graphic of any previous card, the rectangular frame around the animal surrounds all animals in every card because it's in the background. You might also have a common button on the background, one that returns you to the starting point of the stack, for instance. In our figure, we call this the Bkgnd Button (background button).

Figure 2-8 shows a typical set of cards and an associated background for our animals stack. In this diagram, the first card is the one facing you and the last card is the one farthest away. You learn more about this concept when we get into navigation at the end of this chapter. Note that the common elements on the background layer, such as the frame and the background button, appear on all the cards. At this point, the stack is homogeneous—it has only one background.

The Stack

A HyperCard stack is made up of one or more backgrounds and its associated cards, as shown in Figure 2-9. Such a stack shows up on the desktop of your Macintosh as a HyperCard *document icon*. The fields and buttons in the stack are all located on a particular card of a particular background. Now, our stack appears to be heterogeneous: there are three backgrounds, all somewhat alike. In this case, the background identifies larger animal categories: birds, dogs, animals of Asia, etc.

The Rest of the Story

HyperCard stacks don't run without the HyperCard application product in the computer. In fact, there is an intimate flow of information from your stack to the Home stack, and finally to HyperCard, as shown in Figure 2-10. This means that, unlike regular Macintosh applications, HyperCard stacks can't run stand-alone and always require the HyperCard (350K bytes at present) application to be on the computer. This may hold back professional programmers from considering HyperCard and

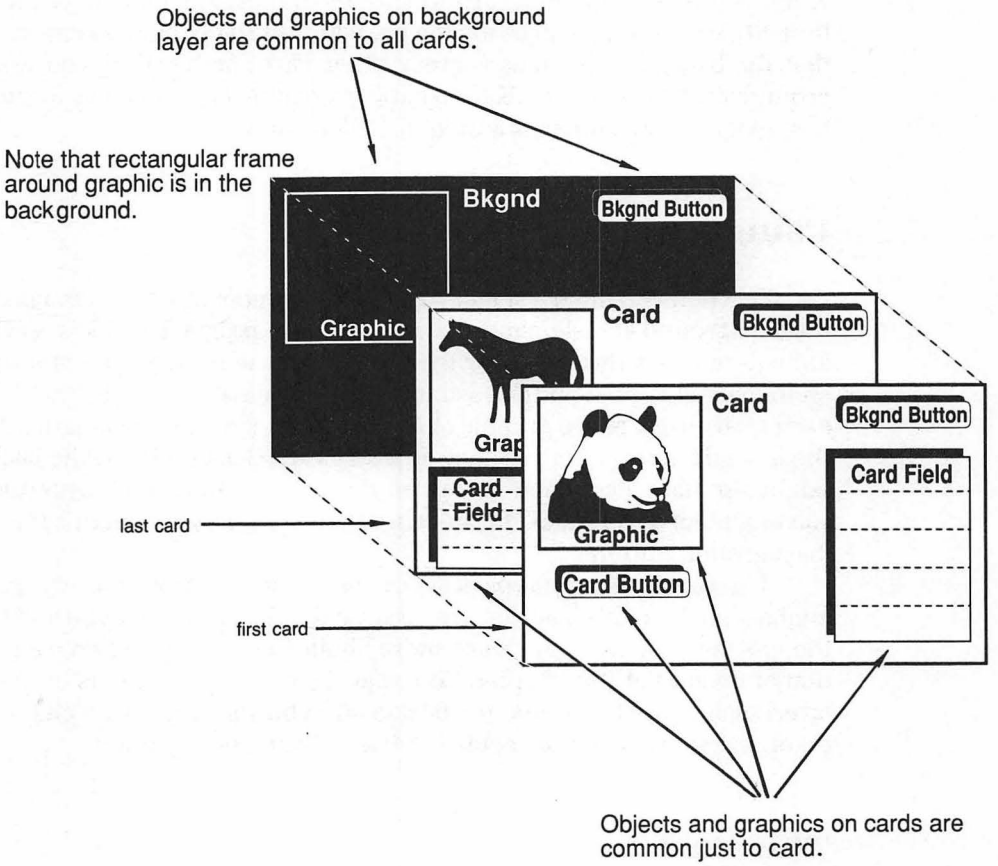


Figure 2-8. Elements of the stack.

HyperTalk as a general solution to programming problems, because it is more complicated to ensure that the user has the correct version of HyperCard. We have found, however, that because the HyperCard product is bundled with each Apple Macintosh computer, the problem is ameliorated somewhat. The ideal solution would be to make a HyperCard compiler that allows turning your stacks into standalone applications.

Stacks

Theoretically, a stack in HyperCard can be up to 512 megabytes. That is larger than most commercial disk drives for the Macintosh computer by quite a few megabytes (typical hard disks for the Macintosh are 20 to 100 megabytes). The smallest a stack can be is 4,896 bytes. Like cards, but-

tons, and fields, a stack script can be up to 30,000 characters and a stack name can be up to 31 characters. You might think that the limit of 16,777,216 cards is enough to handle most situations, but suppose you want to put every person in the U.S.A. on a card. That requires over 250 million cards, so we have to break them into multiple stacks. Suppose we use 25 stacks. That gives us 10 million cards per stack. Because the maximum size of a stack is 512 megabytes and we have 10 million cards, we can put 51 bytes on each card. Obviously, we need fewer cards per stack. Suppose we devote 32K per card to each person. That means we need 512M ÷ 32K, or 16,000 cards per stack. Since we need 250 million people in our database, there are 250M ÷ 16,000, or 15,625 stacks to deal with!

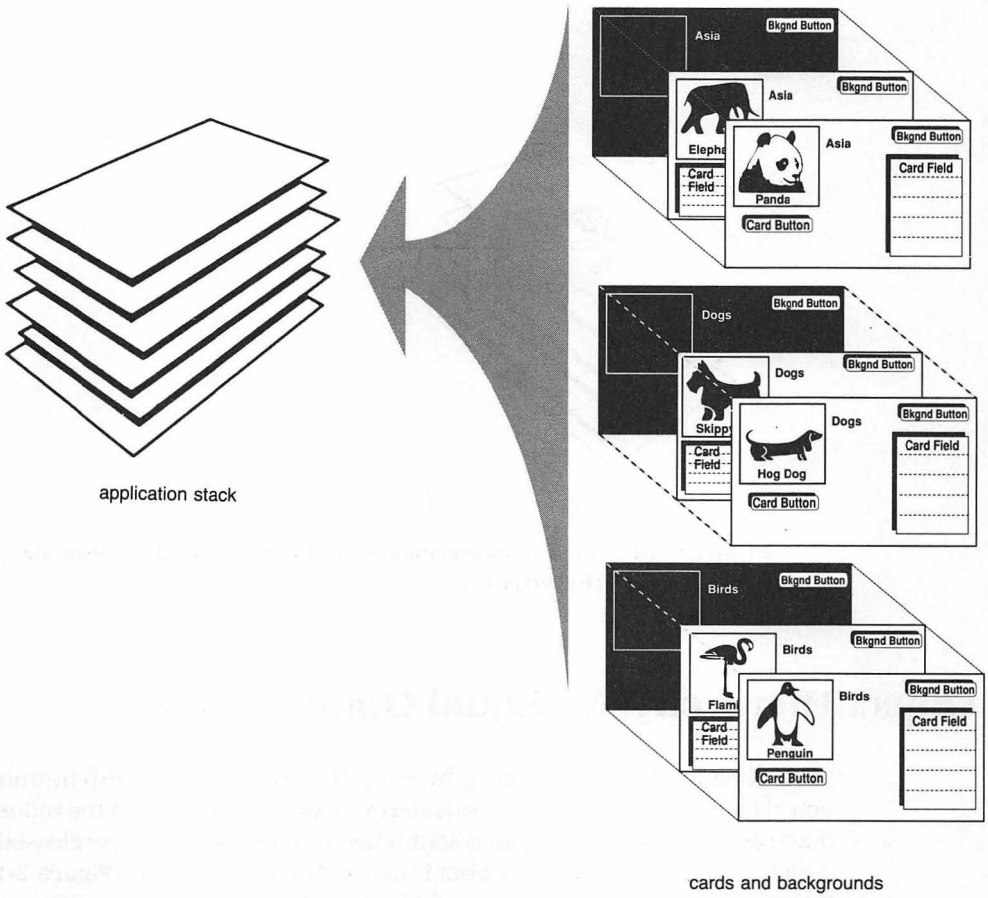


Figure 2-9. Stack universe made from multiple backgrounds and cards.

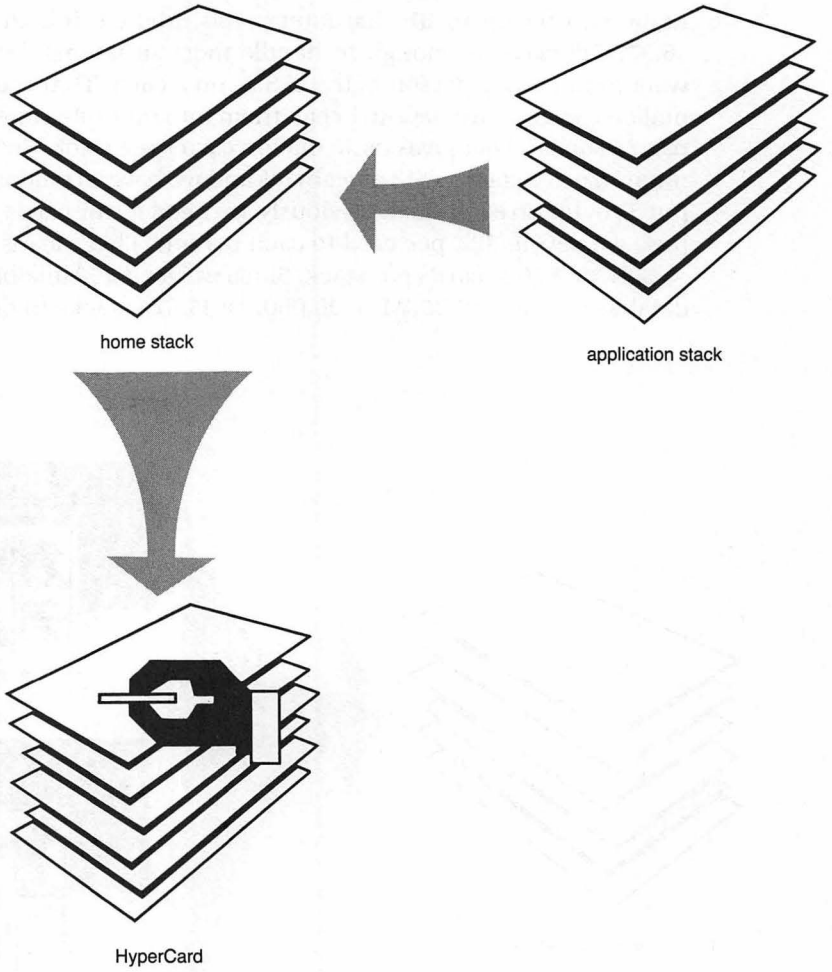


Figure 2-10. Flow of information from application stack to Home stack to HyperCard.

Object Hierarchy: No Equal Opportunity

To understand the object hierarchy, consider a normal card with buttons on it. When you click the mouse button, a `mouseDown` message is sent when the mouse button goes down and a `mouseUp` message is sent when the mouse button is released. The point at which a message enters the object is called its *entry point*. See Figure 2-11. Remember, the script inside the button is called a handler because it traps the message of the mouse being clicked and causes a beep in the speaker, i.e., it "handles" the message. Not of much practical value this button that beeps, but it does illustrate that when the cur-

sor (the browse hand) is over the button and clicked, the `mouseDown` and `mouseUp` messages are sent to it. If there is a handler for that message, it is absorbed or trapped and used up. The message sort of dies, but, at the same time, it triggers the handler, which then does its thing. But what if you click outside the button on a card?

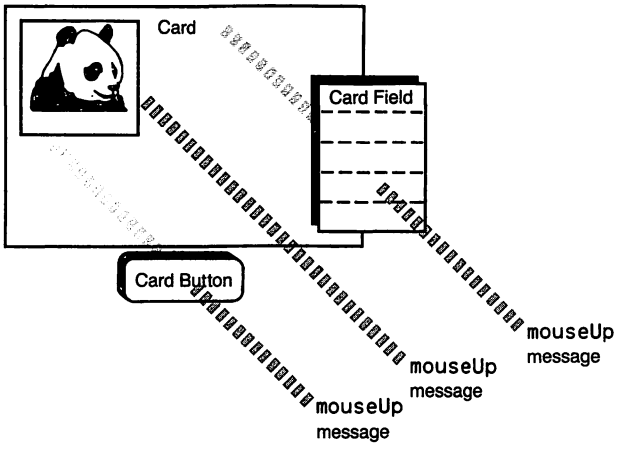


Figure 2-11. Message moves from buttons and field to cards.

In that case, the `mouseDown` and `mouseUp` messages (and other messages you shall learn about later) are sent to the card. Because a card is an object, it can also contain a script, so a mouseDown handler reacts and a beep could be put in it just as we did in the button.

Fields work like buttons as far as messages are concerned. If you click a field, you send a `mouseDown` and `mouseUp` message to it that can be trapped just like a button handler traps, provided the text field is locked. Click outside the field on the card and the same message goes right to the card.

What happens if you don't have a handler in a button or field, and do have one in the card? Does the message go through the button and end up in the card, or is it still blocked from the card?

Look at Figure 2-12, an illustration of object hierarchy, to see how messages flow in HyperCard. If the button doesn't have handlers for the messages, the messages pass automatically on to the next object—in the case of buttons or fields, this is the card. There is no handler in the button or field, so there is nothing to trap the message and the message gets passed. It proceeds to the card layer, and if that object has a handler in it, the message gets trapped there. As far as the messages are concerned, there is an order of precedence being established. Mouse-related messages first go to any buttons and fields they are over, and if these contain no handlers, they pass to the card. If there is no button or field under the cursor, the mouse message goes directly to the card, taking a more direct route.

Messages don't stop here, however. If the card doesn't have handlers for the

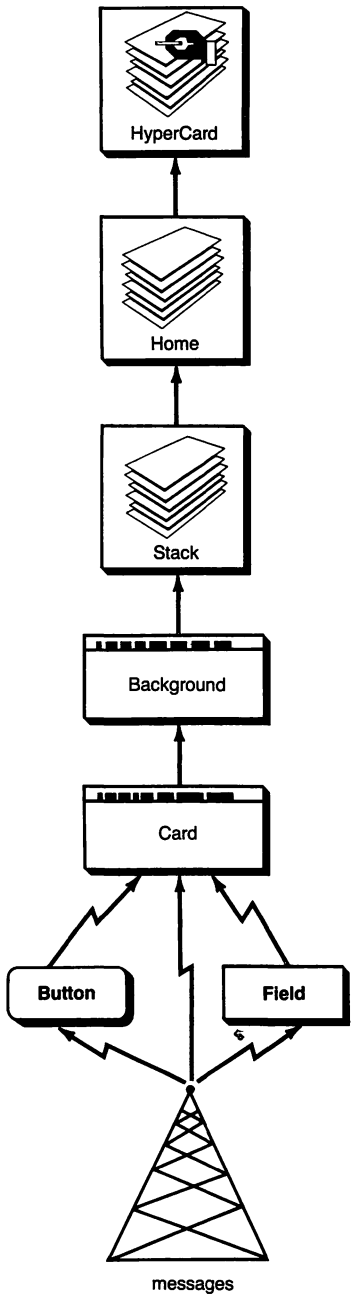


Figure 2-12. Object hierarchy.

messages, they pass on to the background object. If that object doesn't have a handler, the messages pass on to the stack object, then to the Home stack, and finally to HyperCard itself.

Note that messages go directly from a button or field to the card. They can't go to another button or field, even if it's sitting right under the one you clicked first. The only way to make a message go to another button or field is to specifically direct it to the object using the `send` command, which you learn about in chapter 6, "Messages."

The object hierarchy is a fundamental HyperTalk concept. HyperTalk runs on messages, and you need to understand how those messages are flowing to harness efficiently the language's power. The best example of using the hierarchy to your advantage is handler sharing. The higher a handler is placed in the hierarchy, the more widely it is shared because messages can come to it from more objects. To demonstrate this, you can put a mouseDown handler in the Home stack (the top of the hierarchy) which is activated by pressing the mouse down in any button or card in any stack (provided there is no other mouseDown handler at a lower level blocking the passage of the message).

Example 2:

1. Go to your Home stack.
2. Enter the following handler in the stack script. You can put it at the beginning of the existing scripts in the Home stack (don't worry—you remove it later):

```
on mouseDown -- put in your Home stack
  play "boing"
end mouseDown
```

3. Click on the background of your home card.

Result:

You get a boing sound. This shows that the message is traveling to the card of your Home stack, then not being absorbed but instead passing on to the background object. In the background object, the same thing happens—the message is passed because there is no mouseDown handler. Finally, the message gets grabbed in the Home stack object where our play "boing" handler is waiting patiently for a chance to grab the message and then spit out its own audio gasp.

4. Now move to any other stack, such as your HyperTalk Bible stack, and click a card.

Result:

You should hear the same boing.

The message is entering all the layers shown in Figure 2-11, and is being passed by each object because there are no mouseDown handlers in these objects to trap it. The message then moves out of the stack and into your Home stack where it finally finds the mouseDown handler we planted there. If you click on any button, the button's job likely is performed, and if there is no specific mouseDown handler, the boing sounds as well. To just hear the boing without activating the button's script, press down on the button and slide the cursor off the button before releasing the

mouse button. You probably want to delete that handler unless you are a glutton for audio feedback, but the thing to remember is that handlers placed in the Home stack's stack script are accessible as long as you are running off that Home stack. There are many occasions when you want to place a handler in the Home stack rather than duplicate it time and again at lower levels.

Of course, there are other handlers that you want to place as low in the hierarchy as possible because they pertain to only one special case. In the next section, we introduce a new command that lets you have the best of both worlds: low-level handlers for special cases and higher-level handlers for the same message that are applied more generally.

Keep off the Grass

A warning: if you are writing stacks that will be used by other people, you may be tempted to consider installing parts of your scripts in the user's Home stack. Because a handler in the Home stack is accessible from all stacks, tools, utilities, stack space monitors, and the like would seem best placed there. However, be cautious about this. Because it's so easy to modify HyperCard, many people have customized their Home stack, and its contents can't be counted on to be standard. Suppose, for example, you create a handler that automatically compresses stacks when the free space of a stack has grown to greater than 20K, and the user's Home stack already had such a handler. Some strange things may happen. It's also possible that you have used global variables that are already defined in the user's Home stack, or that you are accessing fields, such as the home card time field, that the user may have removed. What is the solution? You might try checking to see whether there is a handler in the user's stack that might be corrupted, or if the action your stack is about to perform works in your particular situation. Try to anticipate the user with the most nonstandard home card using your stack. That minimizes your problems later.

Passing the Message

Normally, a message is used up when it is handled by a script. The message goes in the top of the handler and nothing comes out the other end. There are times when you want to run one handler off a particular message at one level and still have the message continue on to higher objects with other handlers. For example, you might want to have a button handler perform some visual effect while the card handler does the major computing. Or you might want to capture the list of buttons someone has clicked and place an on mouseDown handler in the stack object. You can do this by including the `pass` command at the end of the first handler. The `pass` command syntax is illustrated in Figure 2-13.

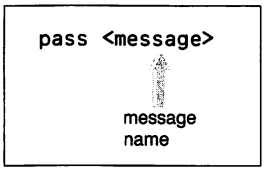


Figure 2-13. The pass command syntax.

The word pass is followed by the name of the message to pass. The statement goes inside your handler in the object you wish to pass the message through after it's handled. You might see these statements:

```
on mouseUp
  --
  -- script that processes here
  --
  pass mouseUp
end mouseUp
```

The effect of the pass command is to add a sort of back door to the handler so the message can pass through it and continue on up the hierarchy. In Figure 2-14, pass can handle the mouse messages, then pass them to the card. Both buttons contain on mouseUp handlers. In the next example, we demonstrate the effect of the pass command at several object levels. This exercise gives you a good understanding of the entire object hierarchy.

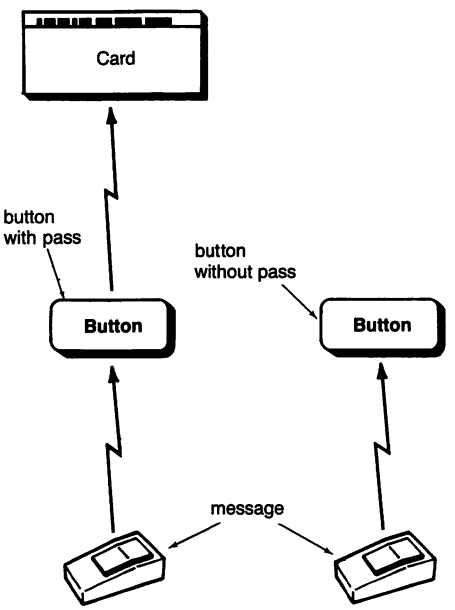


Figure 2-14. Effect of the pass command.

Example 3:

1. If your HyperTalk Bible stack is not already open, open it and go to the Objects card. This should be the second card in the stack, the one with the diamond graphic.
2. Add a new card button to the card. Place it as shown in Figure 2-15 and name it `myButton1`.

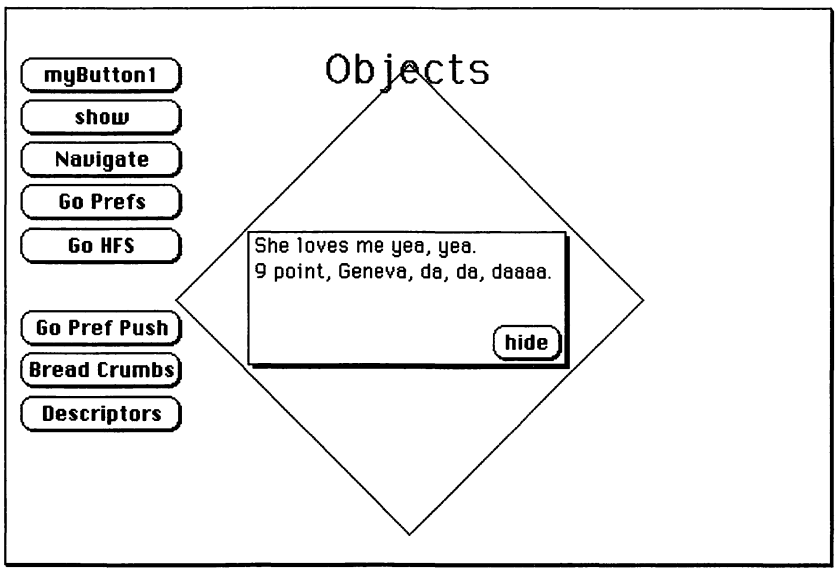


Figure 2-15. The complete Objects card.

There are several buttons and fields on the card in Figure 2-15 other than the mybutton1 button because the figure shows the final state of the card. We do not show the card every time we add a new button or field. As you follow the examples, place your buttons or fields in the locations shown by the figure as if the other parts were not already on the card.

3. Enter the following handler in the myButton1 button script:

```
on mouseUp -- myButton1
  put "myButton1" into msg
  play "harpsichord" 70
  wait 30 ticks
  pass mouseUp
end mouseUp
```

We explain the `wait` command in more detail later, but for now, understand that `wait` causes the script to stop for a certain amount of time. Here, we are using *tick* as the time element, which is equal to 1/60 of a second. So `wait 30 ticks` means wait 30/60 second, or 1/2 second. The script causes the word `myButton1` to appear in the Message box. It then plays a short bink sound, waits

one-half second (so that you have time to read the message that will soon be replaced), and does a `pass mouseUp` to pass the message to the next object.

4. Enter the following handler in the card script:

```
on mouseUp
  put "card" into msg
  play "harpsichord" 75
  wait 30 ticks
  pass mouseUp
end mouseUp
```

5. Click the myButton1 button.

Result:

First, you hear a bink sound as the first handler in the button responds with the `play` statement running. After a 30-tick delay (1/2-second), you hear a second bink, this time a bit higher in frequency than the first.

We used a message to activate two handlers in two different objects. This message-passing can go further.

6. Add the following handler into the background script:

```
on mouseUp
  put "background" into msg
  play "harpsichord" 80
  wait 30 ticks
  pass mouseUp
end mouseUp
```

7. Add the following handler into the stack script:

```
on mouseUp
  put "stack" into msg
  play "harpsichord" 85
  wait 30 ticks
end mouseUp
```

8. Click the myButton1 button.

Result:

You hear the bink sounds, rising in frequency, and the words `myButton1`, `card`, `background`, and `stack` appear in the Message box.

It's important to note that the `pass` command must be at the end of the handler or the statements below it are not executed.

Although the normal way messages flow is vertically from buttons and fields to cards, it is possible to make messages move laterally, or horizontally, from a button to another button or field at the same level or even vertically from Home to a button on a card. This requires the `send` command which is covered in chapter 6, "Messages."

Buttons and Field Levels

One confounding thing you'll find when using messages is that they pass from buttons to cards, or from fields to cards, but not from buttons to fields or fields to buttons. Although you can control the layers that buttons and fields are in so that one is physically on top of another, this layer-

ing does not mean a message will go from a button to the field under it or vice versa. You need to explicitly send a message from one object to another to make that happen.

A Humble Galactic HyperCard Metaphor

With so many different kinds of objects, layers, and messages flowing back and forth, it is likely this whole relationship makes your head spin. We put together a way of visualizing how HyperCard works that might help you understand it better. Figure 2-16 shows how our metaphor starts. In this figure, buttons and fields are shown orbiting around the card. Each button and field is in its own orbit, just as each button and field is in its own layer in HyperCard. If two buttons line up so that a mouse is over both of them, the message goes to the button in the outermost orbit. If there is no impinging button or field between the mouse and the card, the message goes directly to the card.

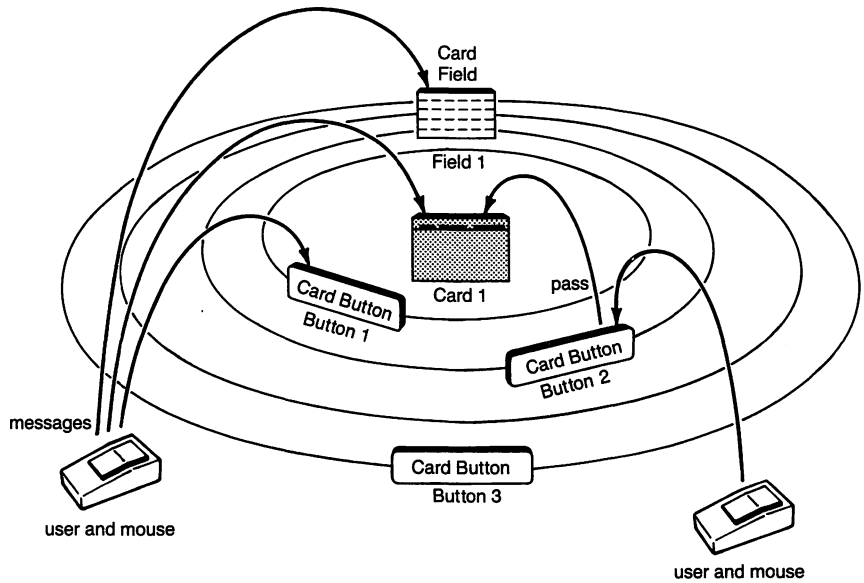


Figure 2-16. Relation between cards, buttons, and fields.

You learned that cards and their associated parts are always attached to a particular background card. Remember, cards can all "see" the background object, but not necessarily the other cards. If you stretch your imagination, the background card might be considered a planet of intelligent beings, and the cards are its satellites (see Figure 2-17). In this particular Hyper-Universe the satellites are all in the

same orbit, and are so far away from each other than one card can't see the other very well. Because a background can have its own parts, you might imagine that buttons and fields can also orbit close into the planet, where all the cards (satellites) can see them.

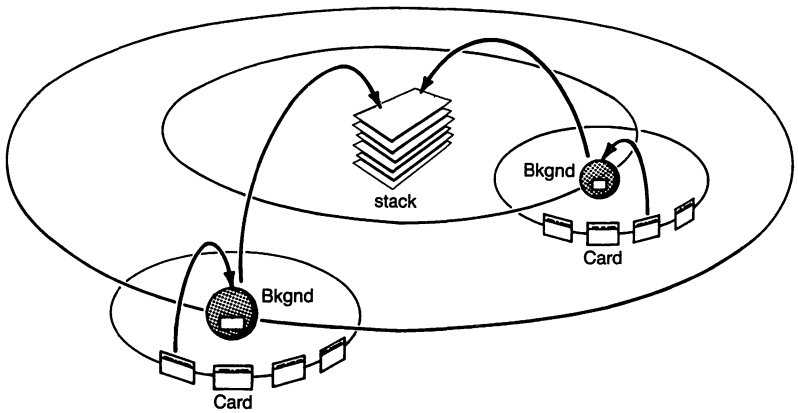


Figure 2-17. Relation between stack, background, and cards.

A collection of backgrounds and their cards makes up a stack. If you are willing to take our analogy another step forward, a stack can be considered a sun and the cards and associated parts orbiting planets, as in Figure 2-17. Messages travel from the cards to the backgrounds to the stack. What we have here is a HyperCard solar system.

Imagine that a collection of stacks is like a collection of suns, so stacks, along with the HyperCard program and the home card, make up the complete HyperCard galaxy. Stacks must orbit HyperCard, because they can't (presently) run without it. This is shown in Figure 2-18. We won't bore you with too many trivial extrapolations of this metaphor, but one irresistible path is this: if HyperCard forms a galaxy, different applications must correspond to different types of galaxies. Therefore, the collection of all applications must form the Macintosh universe. Although intergalactic travel is quite easy in the Macintosh universe (using MultiFinder), there are rumored to be strange, opposite, and hostile universes, where all the interface features of the Macintosh are reversed.

Naming Objects

We have been talking about objects but we have not said how HyperTalk scripts can refer to these objects. HyperCard has an elaborate naming scheme that lets you identify specific buttons, fields, and other objects from your scripts.

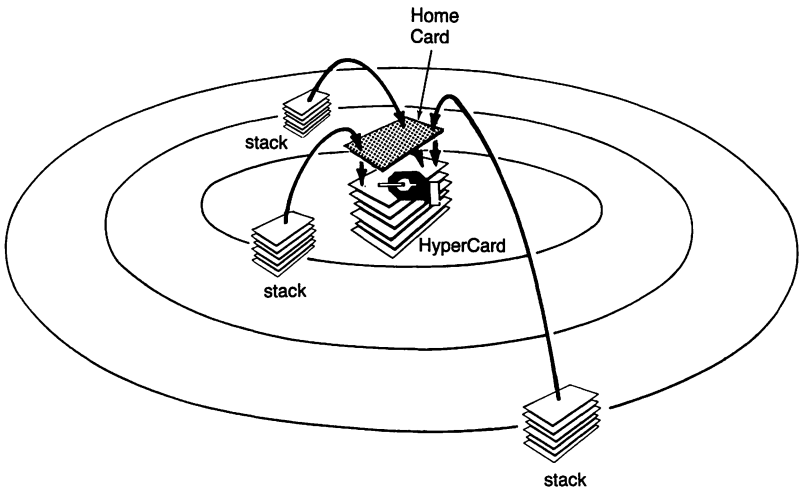


Figure 2-18. Relation between stacks, home card, and HyperCard.

There are three ways to refer to fields, buttons, cards, and backgrounds. You can refer to them by name, the way we named our first two cards of HyperTalk Bible Getting Started or Objects. You can refer to objects by relative number, like `card 1` or `button 10`. Finally, you can refer to objects by a unique identification number, like `field id 4573` or `background id 967`. Stacks are different in that they only have names, but all the other objects can be designated in all three ways.

Names

You, or the user, supply the name of an object by typing it into the Name field in the info dialog box (accessed from the Objects menu or by double-clicking buttons or fields that have been selected), by setting the name property in a script, or by typing the object's name into the Message box. Using unique names for objects has many advantages, as you shall see as you learn HyperTalk, but the main advantage is you can make a label name follow some logic so that it can identify what it actually does. For example, calling a button Prev gives us a pretty good indication that it sends us to the previous card. The syntax for referencing objects by name is illustrated in Figure 2-19.

This syntax shows that we must type the object type first (card, button, or stack), and follow this with the specific name of the object. Some examples of specifying named objects would be `card button "Herbert"`, `background "Hoover"`, or `stack "Presidents"`. Names can be up to 31 characters long, must start with a letter, and can contain letters, numbers, and some punctuation characters. The name can be either a literal string between quotes, like `"Getting Started"` or it can be a container that holds a string. Chapter 4, "Containers and Variables," discusses naming conventions.

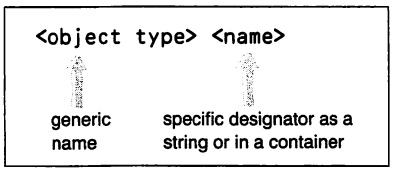


Figure 2-19. Object reference by name.

Relative Numbers

The relative number of an object indicates the position of the object among similar objects. For example, each card in a stack has a relative number starting with one (1), and continuing to the total number of cards of a specified background. Because stacks are not contained in a larger object, they don't have relative numbers.

The syntax for referencing objects by relative numbers has two possible modes, as shown in Figures 2-20 and 2-21. The syntax illustrated in Figure 2-20 is the more common and works like the name designation. We can refer to `card 1`, `button five`, `background 34`, or `field 1`. The syntax in Figure 2-21 refers to objects with the number followed by the object type, so these are legal designations: `first card`, `any button`, `second background`, or `last field`.

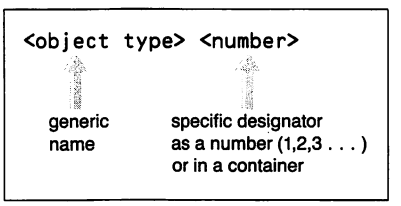


Figure 2-20. Designating objects by relative number.

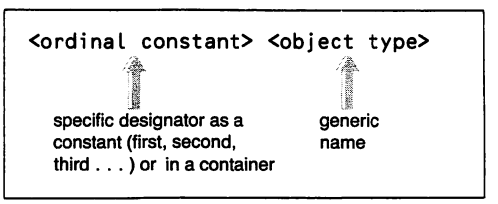


Figure 2-21. Designating objects with the number appearing first.

Card and background buttons and fields have the same relative numbering. On a particular card or background, you have buttons numbered from 1 up to as large as HyperTalk allows (over 32,000 buttons per card). Again, buttons and fields can be accessed with statements like `card button 11`, `background field six`, or `last card field`. The relative number of buttons and fields can be changed with Send Farther and Bring Closer menu commands under the Objects menu.

The value of accessing objects by relative numbers is for stacks where the number of cards and their locations in the stack are fixed. That way, card 19 is always in the same place in the stack, and buttons that take us to them always work. If you sort the stack, add cards, or rearrange the stack, all relative numbers of the cards change, and you have to reprogram all the scripts that were keyed on these relative numbers—an unsightly and dirty task for sure. The relative numbering applies also to buttons and fields.

ID Numbers

When any object is first created, it is assigned a unique and permanent ID number. The object's ID number is like a person's Social Security Number. It's assigned by the powers that be and you are stuck with it from then on.

No similar objects have the same ID in a stack, but similar numbers may be used for similar objects in different stacks.

The syntax for designating objects by ID number is like names—you type the object type, the keyword `id`, and a number, as illustrated in Figure 2-22. An example of referring to an object by ID number is `field id 2`, `background button id 456`, or `background id 13357`.

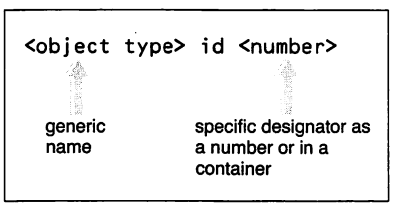


Figure 2-22. Designating an object by ID number.

Remember, HyperTalk itself uses ID numbers when the Link button is used to link the pipe button to the man smoking a pipe card in the Art Stack. The ID numbers are valuable when you want to make sure that an object can be located even when the name is not available. For example, a script that makes cards can store the ID of each card and refer to it later that way. Another advantage to referring to objects by ID number is that the user can't accidentally change the object's ID because it is not typed into a field the way a name is. However, because a user can delete your objects, regardless of how they are named, this value is not that useful.

Object Hide and Seek

Before we get too abstract about HyperCard objects, let's investigate a feature where object type is important. The object's buttons, fields, and windows (like the Message box) have the capability of being hidden from view on a card or background and

then made visible again. This method is similar to the "cloaking" device used by the Klingons in Star Trek, except you need no dilithium crystals to make buttons or fields instantly disappear. You can directly access the visible and invisible property of an object, using the `set` command. A short cut to setting the visible property is to use the `show` and `hide` commands. The syntax for these commands is illustrated in Figure 2-23.

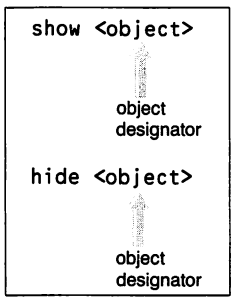


Figure 2-23. The `show` and `hide` command syntaxes.

You can make card field 1 disappear with the statement `hide card field 1`. Likewise, if the object is hidden, you can make it reappear (or pop up) in its last location with the statement `show card field 1`. You can `show` and `hide` card buttons, background buttons, card fields, background fields, and, in Version 1.2 of HyperCard, card pictures and background pictures. Pictures are simply the bit-map painted on the graphics layer of a card.

Showing and hiding objects is one of the key power tools of the HyperTalk scriptor, and something the authoring-level person can't do. For example, with the `show` and `hide` commands, you can create buttons that cause more extensive fields to pop up on a card, where they can be examined. Another button on the field can cause it to be hidden again. Likewise, you can make an entire set of buttons hidden until the user selects a function that makes them visible, and a new section of the screen is created. Showing and hiding fields and buttons can also be exploited to allow animation, as you shall see in chapter 17, "Graphics."

Example 4:

1. Go to the Objects card of the HyperTalk Bible stack.
2. Add a new shadow style card field to the card with New Field, name it `Verse`, and type a famous saying into it.
3. Now bring up the Message box (Command-M), enter the following statement, and press Return:

```
hide card field Verse
```

Result:

The card field named Verse, with its great words of wisdom, will disappear.

4. Double-click the word `hide` in the Message box, replace it with the word `show`, and press the Enter key again. You should be executing the statement

```
show card field Verse
```

Result:

The card field, with its text intact, should reappear where it was.

Pop-Up Note Fields

Perhaps you have already imagined that this technique of showing and hiding fields is key to writing all kinds of interesting stacks. For example, you can make your own pop-up note fields, like the kind that key off a single word surrounded by a transparent button, with just a little more work.

Example 5:

1. Add two new card buttons to your card, name them `show` and `hide`, arrange as shown in Figure 2-15, and enter the following handlers in their scripts:

```
on mouseUp -- show button
  show card field Verse
end mouseUp


on mouseUp -- hide button
  hide card field Verse
end mouseUp
```

   We named each button's script to make it easier to get the right handler in the right button. We will continue this technique whenever there are multiple objects (cards, fields, or buttons) to create in one example.
2. Click the hide button.

Result:

The Verse field disappears.

3. Click the show button.

Result:

The Verse field reappears.

Pop-Ups That Work Better

HyperMedia techniques rely on fields that can be made to pop up when the user clicks on a certain word, graphic symbol, or other custom element. We can make a field visible and invisible, but this two-button show-and-hide affair is not very practical, because you have to clutter the screen with two buttons for every field you wish to pop. There's got to be a better way! How about putting the hide button on the field and hiding it when the field is hidden? That way, when the field pops up, you can read it or enter text in it, and then press the hide button to make it disappear again.

Example 6:

1. Open the show button and add this new statement to it:

```
on mouseUp -- show button
  show card field Verse
  show card button hide -- add this to the show btn
end mouseUp
```

2. Open the hide button and add this new statement to it:

```
on mouseUp -- hide button
  hide card field Verse
  hide card button hide -- add this to the hide btn
end mouseUp
```

3. Now reposition the hide button so it is sitting over the right bottom corner of the field, and resize it as shown in Figure 2-15.

   The hide button must be on top of the field layer. If it is not, and it slides under the field rather than on top of it, you need to move the button forward. With the hide button selected, use the move closer item in the Objects menu, or press the Command-= keys until the button moves over the field layer.
4. Click the hide button.

Result:

The card field and button disappear.

5. Click the show button.

Result:

The card field and the hide button reappear.

We used a name for an object that was also a keyword in the HyperTalk vocabulary (we called the hide button `hide`, and used the command `hide card button hide`). Yet all seemed to work well. This is a feature of HyperTalk not found in other languages. Names of objects can be reserved words. By the way, if you ever hide a part and don't have a script to make it visible again, you can use the Message box to bring it back. All you have to remember is the part's name. For our example, type `show card field Verse` in the Message box and the field appears. Follow this with `show card button hide`.

Using *me*, *target*, and *this*

Although there are great advantages to object-oriented programming, there are also problems, particularly having to refer to objects by name, ID, or relative number. A substantial percentage of the errors made in writing scripts involves misnaming objects. To make this task easier, some additional object-naming features are added to HyperTalk.

Getting to Know *me*

Although it might not be obvious, when we used the statement `hide card button hide`, we did something most computer scientists would consider distasteful. What

could be *that* bad? It is considered poor practice to "hard code" names of any external variable or file into a program. This is also true for naming objects in scripts. If, for example, we decide at some point to change the name of the hide button to `goaway`, we must remember to change its name in the internal script that refers to it, or how could it hide itself anymore? This potential problem can be eliminated in a number of ways, but the simplest is the function called `me`. We haven't explained functions yet (you see them in chapter 5), so for now think of them as simply commands that return answers (a number, text, or name of something). The function `me` replaces itself with the name of the object that the script is inside, the one that initiated the handler in the first place.

For example, if your script is inside a button called hide, using `me` returns the string of letters `card button "hide"`. We then use this string with the `hide` command.

Example 7:

1. Change the script in the hide button as follows:

```
on mouseUp -- hide button
  hide card field Verse
  hide me                    -- add this statement
end mouseUp
```

2. Press the hide button.

Result:

The Verse field and the hide button disappear.

This shows that the statement `hide me` is equivalent to the statement `hide card button hide`. As you will see over and over, creative and clever use of the `me` function leads to simple and intuitive scripts that save you time and energy.

Use *me* Rather than Names

The `me` function allows you to avoid using fixed names and values in scripts that refer to objects. Although the value of this feature might not be apparent at this point, later you will find that using `me` allows you to use the same script in many objects, rather than just the one it was originally designed for. This is called "generalizing" your handlers.

the target of My Affection

Another function that helps with accessing and controlling objects in scripts is called `the target`. This is slightly different than `me`. The function `the target` returns the name of the object that first received the message. We could put the statement `hide the target` into our hide button instead of `hide me`, but that doesn't help us anymore. Suppose you wanted to count the number of times the show and hide buttons were clicked and keep the result in a card layer. (Perhaps you're trying to find out how people are using your stack.) This is where `the target` would work great.

Example 8:

1. Open the card script of your Objects card in the HyperTalk Bible stack.
2. Enter the following handler in it:

```
on mouseUp -- card script
  put the target into the msg box
  play "harpsichord" 75
end mouseUp
```

3. Go to the show and hide buttons and add the statement `pass mouseUp` just before the `end mouseUp` statement. The handlers should look like this now:

```
on mouseUp -- show button
  show card field Verse
  show card button hide
  pass mouseUp            -- add this statement
end mouseUp

on mouseUp -- hide button
  hide card field Verse
  hide me
  pass mouseUp            -- add this statement
end mouseUp
```

4. Click the show button, then click the hide button.

Result:

The Message box displays `card button "show"` when you press the show button, and `card button "hide"` when you press the hide button. If you click the card, the Message box displays `card "objects"` and if you neglect to name the card, it says `card id XXXXX`, where XXXXX is an integer number. A bink sound is played each time you click.

This example should be clear if you understand the message hierarchy. The function `the target` told you the name of the object receiving the first message. When you click the buttons, more than one name appears because the `pass` statement allowed the mouse message to continue to the next object, which was our card. The handler in that card was triggered by the message, and the function `the target` figured out that the original message came from the show or hide button. The function `me` would not help much because it would always return the card name.

Now Hear *this*

In the case of the `me` and `the target` functions, you had to refer to the object that contained your initial message. There is a similar operating command in HyperTalk called `this` that tells more current information about the environment, which you can then use in a script.

The card currently on the screen is `this card`, the background the current card belongs to is `this background`, and the stack the current card is in can be referred to as `this stack`. As with `me`, statements using `this` can be used in handlers

to refer to the current object of the type specified, regardless of which one it is. This card can refer to card 1 or card 34 depending on the situation.

Example 9:

1. Go to the Objects card of your HyperTalk Bible stack.
2. Enter the following command in the Message box:

```
put the name of this stack into msg
```

Result:

`HyperTalk Bible stack`, or whatever the name of your stack is, appears in the Message box.

How might `this` be used? You could add a handler to your card so that whenever you moved to it, its name would be read and placed in a card field for the user. This would be practical if you were building a stack that identified different parts in an engine. Each part would have a name like "Head Gasket 35". If you named each card after a distinct part, when the card was first opened, you could get its name and put it in a field. You could add new cards and know that each time you did it, the name of the card would automatically end up in the field for the user.

What's in a Name?

Names of objects can come in three different sizes: plain, long, or short.

Example 10:

1. Open the Objects card and bring up the Message box.
2. Enter `the name of this card` in the Message box and press Return.

Result:

The Message box displays `card "Objects"`.

3. Now enter `the name of this background` into the Message box and press Return.

Result:

The Message box displays something like `bkgnd id 2748` (we have not named the background yet).

When the object specified does not have a name typed into its dialog box, HyperTalk substitutes its unique ID number, as the example reveals. (Note, we don't have to say `put blah blah into the Message box`. We can just say the `blah blah` because `blah blah` is a property of the object.)

The short name of an object in HyperTalk means just the name without the object type.

Example 11:

1. Open the Objects card and bring up the Message box.
2. Enter `the short name of this card` in the Message box and press Return.

Result:

The Message box displays the word `Objects`, because that is the name of the card. Note that the text card is not included.

The long name of an object is its name with all its type and location descriptions tacked on.

Example 12:

1. Open the Objects card and bring up the Message box.
2. Enter `the long name of this card` in the Message box and press Return.

Result:

The Message box display depends on how you have arranged your stack and folders, but is something like

```
card "Objects" of stack "Draw/Hyper:Hyper Folder:HyperCard
Stacks:HyperTalk Bible"
```

The entire route you would have to give to have someone find that card is spelled out by the long name property. This list corresponds to the name of the location of the card displayed in the document name area above the card and, if it's long, it may not all fit in the first line of the Message box (but it is still there).

Pathnames: Zipcodes for Stacks

Often you wish to control or refer to buttons and fields of cards that are not in the stack you are currently using. Take, for instance, the situation where your application is comprised of several stacks. Perhaps you are using your stack to control the inventory of your company's products and you have a stack called engine, another called drive train, and so on—one stack for each major parts area. In such a case, access the cards in the various stacks from a master stack. You want to get information from those stacks, such as the contents of fields, the names of buttons, or the state of certain pixels.

You have to identify the name of a specific card in addition to the name of the field. That card can be in another stack, and that stack can be in nested folders, on another disk connected to your Macintosh, or on another server or Macintosh on a network. It could be on a CD ROM disk (which can be hundreds of megabytes in size), or even on another computer accessed via a satellite. In all these cases, you have to specify the names of all the HyperCard objects and HFS (Hierarchical File System) entities that lead to your final objective. This chain of names is called the *pathname*. Example 12 showed this accessing method. (Note that you can refer to a button or, more likely, a field on another card in the same stack without having to go to that card. This ability comes in handy in later chapters when we start using fields as containers for values and when we start controlling the flow of messages.)

To refer to containing objects, all you have to do is string together the object

descriptors you've already learned with the words `of` or `in` between them, as illustrated in Figure 2-24.

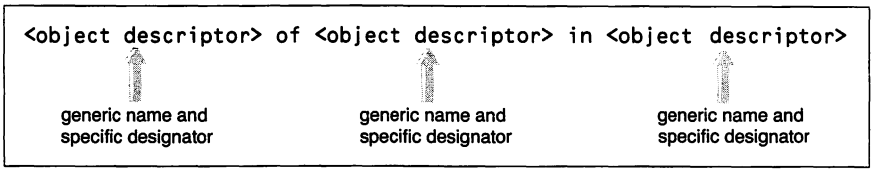


Figure 2-24. Concatenated object descriptor syntax.

For example, if you want to hide a button called Special Editor Mode (because it has been determined that the user is a nonexpert), located on the background card called Carburetor, in the Engine stack of our inventory collection, you could write it as `hide card button "Special Editor Mode" of background "Carburetor" of stack "Engine"`.

Another example is `card button 1 of card id 2345 of stack "Home"`. The object descriptors are read from left to right starting with the smaller object (fields and buttons) and moving up to the largest (backgrounds, stacks, home card, HyperCard). As long as the stack referred to is in the current folder or is in a folder HyperCard knows the path to (the pathname is stored on card 2 of the Home stack), HyperCard will find the object.

Another neat feature of HyperCard (and HyperTalk) is its preemptive user assistance. Even if HyperCard doesn't know the path, or you fail to give it one with the `go` command, the user is presented a standard files dialog box and is permitted to click through the folders to the desired stack.

When HyperCard doesn't already know the HFS path and you don't want to click through the open box, you have to explicitly provide the information in the same way HyperTalk provided it when we used the long name. The format is `"DiskName:OuterFolder:InnerFolder: StackName"`. For more information on pathname, see chapter 19, "File Input and Output."

Navigation

As a HyperTalk programmer you are the Wizard of Oz. You stand hidden behind your curtain of celebrated reputation while your scripts control what stacks, pictures, cards, and fields users, like Dorothy, the Scarecrow, the Lion, and the Tin Man, can see. Consequently, how you move the user through your stack becomes the basic "feel" of your application. Do a poor job of moving between cards, present boring pictures, confuse the user with illogical jumps from one card to another, bore the user with long delays as the application moves among so many different stacks, and chances are that you, the Wizard, will be exposed for what you are: a big showoff. The user probably won't continue using your stack. Do a good job of mov-

ing between cards, use nice special effects and sounds, make the jumps between cards logical and easy to follow. Then, assuming the "content" is interesting, this smooth movement increases the chances of our foursome having great respect and admiration for you, the Wizard behind the scenes, and increases the odds of your stack's being used more than once.

Moving around among cards and stacks is called *navigation* and is performed in scripts with the aid of three Hypertalk commands: `go`, `push`, and `pop`. Now that you have a good foundation in object-naming, we take you for a spin with these navigational commands.

go for It

All you need to go from one card to any other card in any other stack is the `go` command and a way of referring to the card you want to go to. The syntax for `go` is illustrated in Figure 2-25.

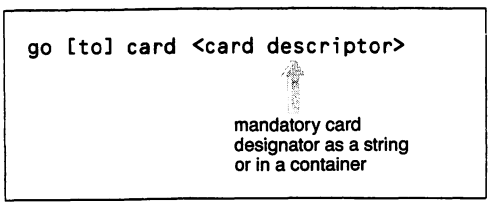


Figure 2-25. The `go` command syntax.

As you can see, the word `to` is optional, so you can say `go to card Mitch`, or just `go card Mitch`. The card descriptor can be a literal value like `go card 1`, or `go card "Objects"`, or it can be in a container, like `go to card card field 1`, which takes you to the card whose name is defined in `card field 1`.

To navigate from one card to another within the same stack, all you need to do is give the name of the destination card object. This can be a name between quotes or a container holding that name. With the `go` command, the destination stack is assumed to be the current stack unless you indicate otherwise.

Example 13:

1. Go to the Objects card in your HyperTalk Bible stack.
2. Create a new button, name it `Navigate`, and enter the following script in it:

```
on mouseUp -- Navigate button
  play "harpsichord" 80
  go card 1                -- here is the first go
  wait 30 ticks
  play "harpsichord" 70
  go card "Objects"        -- here is the second go
end mouseUp
```

Result:

You hear a bink sound, the screen switches to the first card, waits a half-second, plays another bink, and goes back to the second card.

This example shows how we can go to a card by relative number (`go card 1`) or by using the name (`go card "Objects"`).

go to card* Is Like *go to subroutine

It's easy for BASIC programmers to look at HyperTalk's `go to card` statement and equate it to BASIC's `goto` statement. However, they are only superficially equivalent. In BASIC, a `goto` gets you to a body of code to execute. When you `go to` a card in HyperTalk, you cause processing to occur if you add a handler in the card called on openCard. The overall effect in HyperCard is to move you to a new object.

Going to a card in another stack is not much harder, you just have to add the name of the destination stack: `go to last card of stack "Home"`. In fact, if you just want to go to the first card of a stack, you don't need to include a card designator at all: `go stack "Home"` takes you to the first card of the Home stack. The syntax is illustrated in Figure 2-26. As you can see, the words to and of stack are optional.

```
go [to] [card <card descriptor>] [of stack] <stack descriptor>
```

optional card designator as a string or in a container

mandatory stack name as a string or in a container

Figure 2-26. The `go` command to another stack.

Suppose you want to allow the user to change the Preferences card in the Home stack from the current stack. We don't know for sure whether the user has added new cards to the Home stack, so we use the name of the card as our descriptor.

Example 14:

1. Open the Objects card in your HyperTalk Bible stack.
2. Add a new card button to the card and name it `Go Prefs`.
3. Enter the following handler in the button:

```
on mouseUp -- Go Prefs
  go card "User Preferences" of "home"
  wait 1 seconds
  go to card "Objects" of stack "HyperTalk Bible"
end mouseUp
```

4. Click the go user pref button.

Result:

HyperCard goes to the Preferences card, waits one second, then immediately returns to the Objects card.

If your HyperTalk Bible stack is not in the same folder as your Home stack, this script does not know its location. The handler was still running when the screen showed the new card, and that is how it returned. Had we left the second `go` out, our user would have been relocated to the Home stack. You can go to any card in any stack and restart the user's as if it were in the same stack. We are assuming that the HyperTalk Bible stack is in the same folder as HyperCard's home card.

There are stacks that require an HFS trek to get to. In the next example, we ask you to quit HyperCard for the Finder and make some new folders. If you have a desk accessory that allows you to create folders from inside an application, or if you are using MultiFinder, you may want to alter the following instructions accordingly.

The following example requires that you make a new folder and move a stack into it and that you have your folders set up much like ours. Because everyone sets up folders differently, you may not wish to actually try this unless you have your system set up like ours. You could modify your pathnames so that they correspond to your own arrangement, or modify the pathnames used in the example to match your situation.

Example 15:

1. Quit HyperCard. Make a new folder and name it `myApp Folder`. Move the Address stack that comes with HyperCard into this folder. If you don't have such a stack, make a new stack, name it `Address`, and put it in the myApp Folder.
2. Start HyperCard, go to the Objects card of your HyperTalk Bible stack, and create a new button. Name the button `Go HFS`.

   We assume your system is set up so that the folder Hyper Folder contains the HyperCard application, the folder HyperCard Stacks, and the folder myApp Folder. Inside the HyperCard Stacks folder is the Home stack and the HyperTalk Bible stack. Within the myApp Folder is the Address stack. The name of our disk drive is Draw/Hyper.
3. Enter the following handler in the Go HFS button:

```
on mouseUp -- Go HFS button
  go to stack "Draw/Hyper:Hyper Folder:MyApp Folder:Address"
  wait 1 seconds
  go to card "Objects" of¬
  stack "Draw/Hyper:Hyper Folder:HyperCard Stacks:HyperTalk Bible"
end mouseUp
```

4. Click the Go HFS button.

Result:

The first address card in the Address stack is displayed for one second, then you are returned to the original Objects card.

We needed to provide a pathname when we returned to the original stack because HyperCard didn't remember the path. You can't put the option-return character into the pathname (or between the quotes).

When *push* Comes to *pop*

We left our HyperTalk Bible stack a number of times and then returned with a `go` command. There is a better way to do this. The command `push card` causes HyperCard to remember the current card so that it can return to it at a later time. The command that causes the return is called `pop card`. This makes it easy for us to code a script to do a return to the caller card without putting a hard name into it. For example, in our card called go user pref, we had to hard code the return statement as `go to card "Objects" of stack "HyperTalk Bible stack"`. We can make this more general with `push` and `pop` because we can eliminate the need to change the names in the script if we change the card that we return to.

Example 16:

1. Open the Objects card in your HyperTalk Bible stack.
2. Copy the go Prefs button, rename it `go Prefs Push`, and enter the following script:

```
on mouseUp -- Go Prefs Push button
  push card
  go card "User Preferences" of "home"
  wait 1 seconds
  pop card
end mouseUp
```

3. Click the button.

Result:

The result is the same as in the previous example, i.e., you go to the Preferences card, then return to the Objects card.

This example shows that `push` saved the card's complete pathname, and `pop` acted like a `go to card path` statement. Get used to using `push` and `pop` often when you are navigating the stack from one card to another.

You can also use multiple `push card` commands to make a bread crumb trail through your stacks and then retrace your steps with the `pop card` command.

Example 17:

1. Open the Objects card in your HyperTalk Bible stack.
2. Add a new card button to the card and name it `Bread Crumbs`.
3. Enter the following handler in the Bread Crumbs button:

```
on mouseUp -- Bread Crumbs button
  push card
  go to card "User Preferences" of stack "Home"
  push card
  go to stack "phone"
  pop card
  pop card
end mouseUp
```

4. Click the Bread Crumbs button.

Result:

HyperCard goes to the Preferences card of the Home stack, then to the phone stack, and returns backward through the same cards. The current location is not coded in the script.

Special Descriptors (Ordinals)

There are a group of special descriptors that are used in the same way numbers are used. These descriptors are: `first`, `middle (mid)`, `last`, `any`, and the functions `this` and `me`. We have already covered `this` and `me` in an earlier section of this chapter. You can think of `first`, `middle`, `last`, and `any` as numbers that specify the first object in a sequence (first card), the middle object (middle card), the last object (last field), or a randomly selected object (any button), respectively. HyperTalk calls these four special descriptors ordinals because they represent an ordering of values.

There are two more descriptors that can be used to aid navigation: `next` and `prev` (previous). The `next card` is the card that follows the current card, the card with the next relative number. The `prev card` refers to the card before the current card.

Example 18:

1. Open the Objects card in your HyperTalk Bible stack.
2. Add a new card button to the card and name it `Descriptors`.
3. Enter the following handler in the new button:

```
on mouseUp -- Special button handler
  push card
  go to first card of "Home"
  wait 30 ticks
  go to last card
  wait 30 ticks
  go to mid card
  wait 30 ticks
  go to any card
  wait 30 ticks
  go to next card
  wait 30 ticks
  go to any card
  wait 30 ticks
  go to prev card
  wait 30 ticks
  pop card
end mouseUp
```

4. Click the Special button.

Result:

HyperCard moves about the cards of the Home stack (which you should understand) and eventually returns to the Objects card in the HyperTalk Bible stack.

There is also the command `go back`, which takes you to the previous card.

Summary

This has been quite a challenging chapter. We refreshed your understanding of the parts that make up the HyperCard universe, because that understanding enables you to manipulate the objects with scripts. You learned the rather large limits to the number of cards in a stack. You learned how the HyperCard object hierarchy is organized and where exactly to put your scripts. Messages, a cornerstone of understanding HyperTalk, were covered and the interesting `pass` command showed you how to let these messages flow through multiple objects.

Naming objects, a rather dull and boring topic, was covered in lots of detail, because HyperCard is not psychic and does not know what you want to do unless you tell it. Accessing objects by relative number, unique ID, and name in scripts was revealed. We gave you a nice HyperTalk script that makes a pop-up field with only a few lines. The options for accessing names of objects—`me`, `the target`, and `this`—were taught. A fuller treatment of names in HyperTalk was given so that you could access HyperTalk's Zip Codes: pathnames. Finally, we learned about navigating among cards and stacks with the `go to` command.

Review Questions

1. True or false? A HyperCard card is just like a regular Macintosh window, except that it's missing scroll bars.
2. What is the maximum number of cards HyperTalk theoretically allows?

   a. 1,024
   b. 1 million
   c. limited to the amount of available RAM
   d. 16,777,216

3. How many buttons can a single card hold?

   a. 30,000
   b. 16,384
   c. 32,767 (However, it is impractical to manipulate this theoretical limit.)
   d. depends on whether MultiFinder or the Finder are used

4. What is the largest amount of text that can be put in a regular HyperCard field?

   a. 30,000 characters
   b. 32,767 characters
   c. depends on font used
   d. 16,384 characters

5. If you completely fill a Macintosh card with black pixels, it uses up about 22 kilobytes. How many bytes on the disk will it consume?

   a. 30 times less
   b. 10 times less

c. 20 times more
d. 5 times more

6. Which of the following describes a heterogeneous stack?

   a. has no more than five backgrounds
   b. has more than one background but each background is the same
   c. has multiple backgrounds
   d. has only one background

7. True or false? A button on a card can be seen by all the card layers above it.
8. A stack can be as large as

   a. 1 gigabyte
   b. 800 megabytes
   c. 800 kilobytes
   d. 512 megabytes

9. A HyperCard message is

   a. a way to use the modem to contact Apple
   b. a signal from a mouse, keyboard, or system event that flows up the HyperCard object hierarchy
   c. a postcard to Bill Atkinson
   d. a subroutine call

10. A handler

    a. holds various scripts until HyperCard is finished
    b. traps a message and executes a script
    c. goes on the end of an object
    d. helps in opening the script editor

11. Which of the following best describe how messages flow in HyperCard?

    a. Messages flow in waves, about one every two minutes.
    b. Messages come in bunches, like people queuing up in a grocery store.
    c. Messages occur as they do in a telephone network, and objects give the equivalent of busy signals.
    d. Messages flow from the message entry point up an object hierarchy, starting with buttons and fields and ending in HyperCard itself.

12. True or false? It's a good idea to put handlers in the user's Home stack.
13. The `pass` command

    a. prevents handlers from absorbing messages
    b. passes the message on to the next lower object in the hierarchy
    c. favors one script over another
    d. shrinks HyperCard's memory requirements

14. The statement `wait 30 ticks`

    a. causes HyperTalk to stop for 30 seconds
    b. causes HyperTalk to stop for 1/2-second
    c. causes HyperTalk to search for 30 ticks (ticks are bugs)
    d. causes HyperTalk to wait for 30 keys to be pushed before continuing

15. True or false? Messages can only flow up the object hierarchy.
16. Objects in HyperCard can be accessed by

    a. name, ID number, and color
    b. name, address, phone number
    c. name, ID number, relative number
    d. relative address, phone number, and family coat of arms

17. Accessing objects by name is superior to accessing them by relative number because

    a. the relative number of a card can change if the position of a card changes, but a name won't
    b. the relative number of a card is too hard to read
    c. the name of a card identifies what the card does

18. ID numbers are a good way to refer to cards when

    a. the name of a card doesn't matter
    b. you wish to make it impossible for the user to rename a card that could damage your stack's structure
    c. you are afraid the card could be lost

19. True or false? You can show and hide card buttons but not card fields.
20. When the object receiving the message is the same one containing the handler for that object, the easy way to access an object name is to

    a. use `me`
    b. use `the target`
    c. use `this card`

21. When the handler for the object is not in the object but is above it in the object hierarchy, the best way to access an object's name is to

    a. use `me`
    b. use `the target`
    c. use `this object`
    d. use `hide`

22. When you want more information about the current card, background, or stack,

    a. use `me`
    b. use `the target`
    c. use `this object`
    d. use `hide`

23. How many ways can you express an object name in HyperTalk?

    a. plain, long, short
    b. plain, regular, long
    c. simple, complex, huge

24. True or false? Pathnames in HyperCard are the links between the various cards, buttons, and fields in a stack.
25. To move between cards from a script, use the

   a. `gosub` command
   b. `go card` command
   c. `goto card` command
   d. `jump` command

26. Which is easier: using `push` and `pop` to navigate between cards, or using hard coded names in the `go` command?
27. Which of the following special designators are not legal?

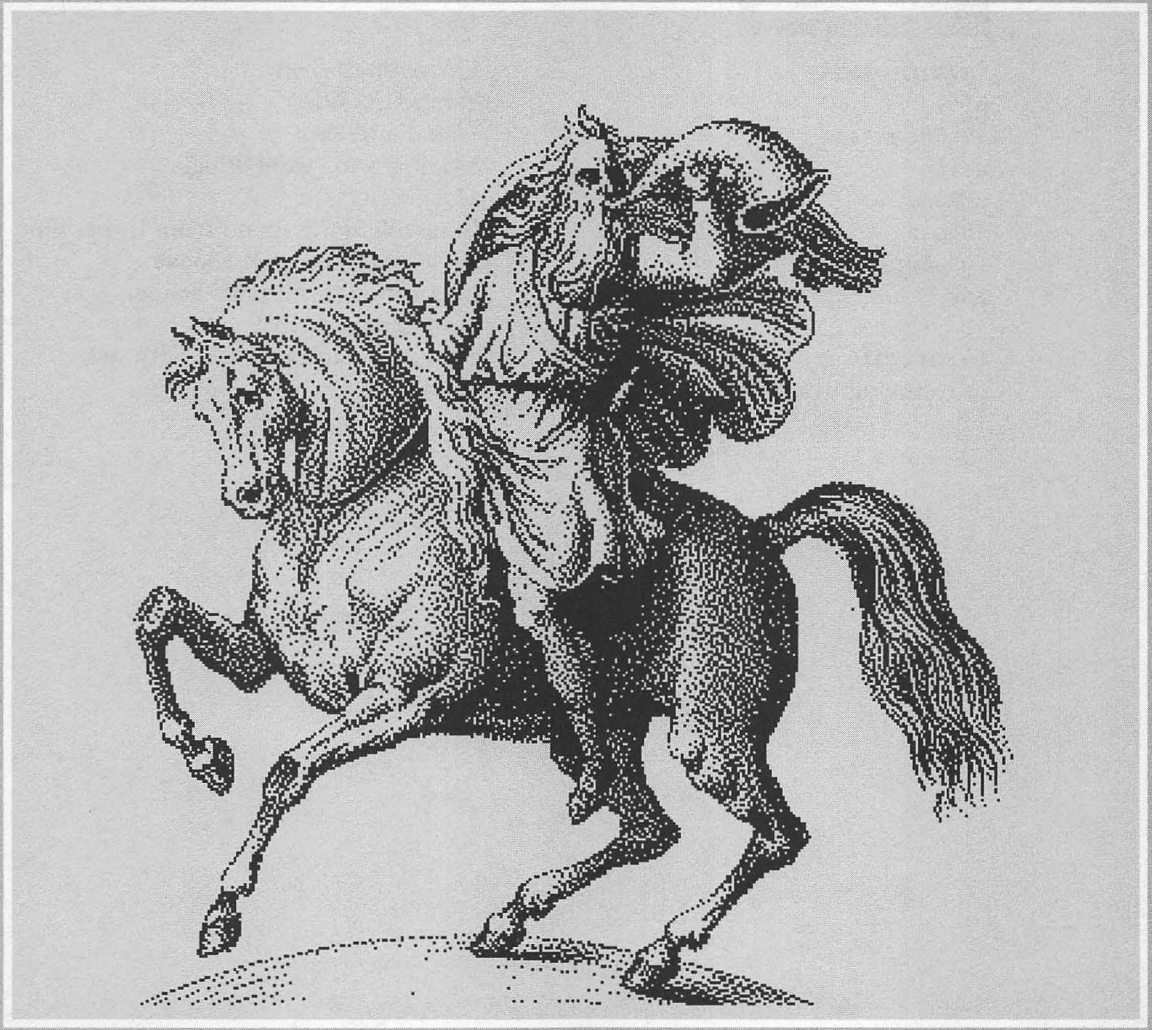
   a. `last`
   b. `first`
   c. `mid`
   d. `any`
   e. `this`
   f. `some`
   g. `prev`

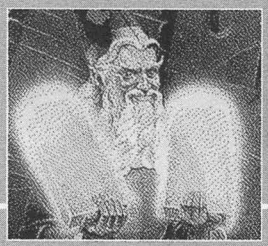
Exercises

1. Write a handler that puts the text `"mouseUp in card"` in the Message box when you click the Objects card.
2. Write a handler that puts the text `"mouseUp in card"` in the Message box when you click the Objects card and a second handler in the background that puts the text `"mouseUp in bkgnd"`.
3. Write a script that puts the long name of a button into the Message box when it's clicked.
4. Write a field script that goes to a random card whenever you click it.

Chapter 3

Audio and Visual Effects





Quick View

Commands

`play`
`visual effect`
`wait`
`doMenu`
`choose`
`answer`
`ask`

Concepts

Introducing `play`

Concepts *cont'd*

Tempo modifier
Visual effects
Modifiers to visual effects
Movie effects
Controlling the menu from HyperTalk
Selecting tools with `choose`
Requesting input with `answer`
Simple delays with `wait`
Requesting information with `ask`

Chapter 3

Audio and Visual Effects

With most computer languages other than HyperTalk, you have to learn a lot of basics before you can do anything interesting. HyperTalk was designed to make simple visual and audio effects accessible to even the most inexperienced scriptor. In this chapter, we show how you can use the `play` command to generate more interesting sounds than the standard Macintosh beep. We also show how you can add visual effects to your card transitions to make going from one card to another a visual event.

In addition, we preview some valuable new commands in this chapter that are used in the chapters that follow. These commands are `doMenu` and `choose`, which let you simulate selecting menu items and tools from inside your scripts and the `answer` and `ask` commands that allow you to display modal dialog boxes. These dialog boxes let you pass information to the user or get information from the user in the usual Macintosh way. We also introduce one version of the `wait` command, which lets you suspend program flow for a specified period of time.

By the end of this chapter, you will be able to create stacks that look better and sound better. Users will be able to interact with your scripts, and your scripts will simulate HyperCard tasks like choosing an item from a menu.

Let's Play: Introducing the *play* Command

HyperTalk's `play` command can be used as a substitute for the `beep` command, or it can be used to actually play music. The `play` command can be used to provide audio feedback for card transitions, for button clicks, or for any other screen activity. The syntax for `play` is shown in Figure 3-1.

(For a discussion of strings and values, see chapter 4, "Containers and Values.")

The `voice` parameter controls the kind of sound generated. If you think of the Macintosh running HyperCard as a disguised music synthesizer, the voice could be a fiddle sound, a flute sound, a piano sound, or whatever voice resource is available. Indeed, the voice can even be a real voice—a person saying "erk," a chipmunk chirruping, anything. Three voices come with HyperCard: boing, harpsichord, and si-

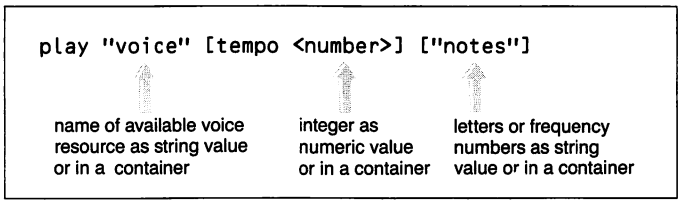


Figure 3-1. The `play` command syntax.

lence. Other voices are available commercially or from BBSs (bulletin board systems) and user groups. For that matter, you can make your own voices with Faralon's MacRecorder (as shown in chapter 15, "Advanced Sound").

A simple `play "boing"` or `play "harpsichord"` command can be used in place of a beep. To make more interesting sounds, you need multiple notes. The range of notes in one octave, starting with the lowest, is C D E F G A B. These notes correspond with the familiar do, re, mi, fa, so, la, and te.

Example 1:

1. Type the following statement into the Message box:

```
play "harpsichord" "c d e f"
```

2. Press the Return or Enter key.

Result:

The notes do re me fa are played.

There is a way to specify notes other than with the letters A through G. Notes can be specified with a number. The four notes in the last exercise can be replaced by the numbers 60 62 64 66. These notes belong to the middle octave, which you will learn in chapter 15, "Advanced Sound," to refer to as *octave 4*. The even numbers are the notes represented by letters and the odd numbers are the sharps and flats between notes (C = 60, C sharp and D flat = 61, and D = 62). Designating a note with a number allows you to specify the exact note in its octave with a single step. Numbers between 50 and 100 sound best, but you can go both below and above this range for special effects.

Example 2:

1. Go to the Objects card in your HyperTalk Bible stack (see chapter 1, "Starting," to learn how to create this stack).
2. Select New Card from the Edit menu, select Card Info... from the Objects menu, and name the new card `Audio-Visual`.
3. Select New Button from the Objects menu, select Button Info... from the Objects menu, and name the new button `Play Numbers`.
4. Replace the mouseUp handler in the Play Numbers button with the following:

```
on mouseUp -- Play Numbers button
  play "harpsichord" "18 20 22 24 26 28 30 32" -- octave 1
```

continued

```
  play "harpsichord" "46 48 50 52 54 56 58 60" -- octave 3
  play "harpsichord" "74 76 78 80 82 84 86 88" -- octave 5
  play "harpsichord" "102 104 106 108 110 112 114 116" -- octave 7
end mouseUp
```

5. Click the Ok button to close the Script Editor, choose the browse tool from the Tool menu, and click the Play Numbers button.

Result:

Plays the scale for four octaves spanning a full seven octaves.

As you can hear, the middle octaves sound best on the Macintosh speaker but the higher and lower notes can be used for special sound effects. The lower notes can be motors or insects and the higher notes make nice clicks when you want cards to snap open or when you want a button to pop into action.

The *play tempo* Modifier

The `play` command has a tempo modifier that controls the speed at which the notes are played. The greater the tempo value, the shorter the interval between notes. (Note that the duration of a note is affected by the pitch. High notes are shorter than low notes.) The default tempo is 120 (120 beats per minute), with higher numbers resulting in faster playing and lower numbers resulting in slower playing.

Example 3:

1. Go to the Audio-Visual card in your HyperTalk Bible stack.
2. Add a new card button and name it `Play Tempo`.
3. Replace the mouseUp handler in the Play Tempo button with the following:

```
on mouseUp -- Play Tempo button
  play "harpsichord" "60 61 62 63 64 65 66"
  play "harpsichord" tempo 700 "60 61 62 63 64 65 66"
  play "harpsichord" tempo 50 "60 61 62 63 64 65 66"
  put "Finished" into the message box
end mouseUp
```

4. Click the Play Tempo button.

Result:

Plays middle C to F notes, including sharps/flats between notes, at the default tempo, repeats them very fast, and finally repeats them again slowly. `Finished` appears in the Message box just after the first note is played and remains while the other notes are played.

It usually takes longer to play musical notes than it takes to execute a HyperTalk statement. As a result, the last line of the handler *appears* to be executed before the middle lines. In fact, the three `play` command statements are sent to HyperCard, where the second and third commands wait for their turns as the first command is playing. Because of this delay between the execution of a statement and

the actual performance of the command, commands mixed in with `play` statements may appear to the user to be executing in a different order than that in which they are listed. (Very brief notes played at a fast tempo are less likely to have this problem. A version of the `wait` command covered in chapter 10, "Repeats," can be used to avoid this confusion.)

You can add `play` sounds to any HyperCard activity—from card transitions to button clicks—but remember that sometimes less is more. Too many sound effects can detract from the real purpose of a stack. Of course, the great thing about HyperTalk is that you can personalize your stacks today, and then tone them down tomorrow when you realize you don't need audio feedback for every action.

Don't Overuse Sound Effects
Keep in mind that HyperCard users may be inundated with the boing and harpsichord sounds. If you make all your buttons boing rather than beep, you won't be doing anyone a favor. A nice substitute for boing and beep is `play "harpsichord" 75`, which makes a soft bink sound.

Visual Effects

A common action in HyperCard is navigating from card to card. One of the easiest enhancements that you can make to your stacks is adding visual effects to those navigational transitions. The HyperTalk command that creates visual effects is called `visual`. Although `visual` can be used without the optional word `effect`, for the sake of clarity, we refer to the command as the `visual effect` command in this chapter. Note that these HyperTalk visual effects always involve the entire card and never just a portion of a card.

In films, special effects let the viewer know that a transition is being made to another place or time. You may recall the iris closing in the James Bond movies or the barn door effects on "Batman."

HyperTalk's visual effects can also provide visual clues to the user. They can signal that the user is viewing the card to the right or the card above the previous card, or they can signal that the user is viewing a card that represents a more profound (enlarged) or a more superficial view (reduced) of the data. Some effects can smooth transitions and others can emphasize them. A list of effects follows.

The Visual Effects
barn door close, barn door open
checkerboard
dissolve
iris close, iris open
plain
scroll down, scroll left, scroll right, scroll up
venetian blinds
wipe down, wipe left, wipe right, wipe up
zoom close, zoom in, zoom open, zoom out

Speed Modifiers
fast, very fast, slow, slowly, very slow, very slowly

Image Modifiers
black, card, gray, inverse, white

Visual effects are fun and easy. The `visual effect` command's syntax is shown in Figure 3-2.

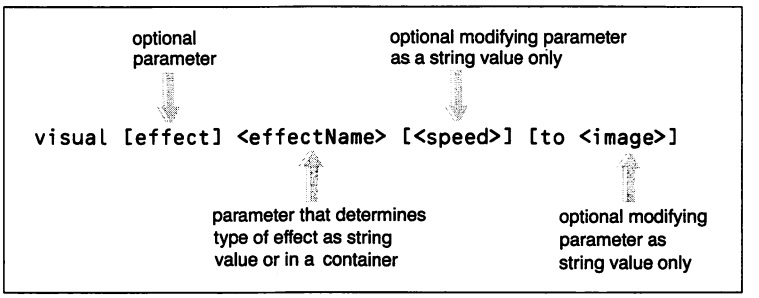


Figure 3-2. The `visual effect` command syntax.

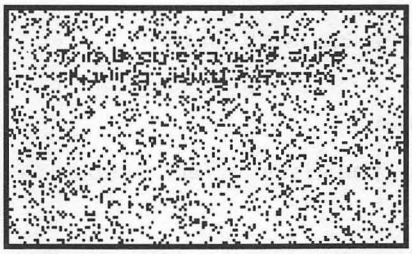
Don't be daunted by the seeming complexity of the `visual effect` command. As is often the case with HyperTalk, there are many options that can be ignored at first while you become familiar with the simpler versions of the command. This HyperTalk command is unusual in that it is only executed if it is followed by a `go` command. Actually, you can have several visual effects run in sequence, but for this to work, a `go` command must appear before the end of the handler, or all the visuals are ignored. The next example shows a very simple visual effect.

The Dissolve Effect

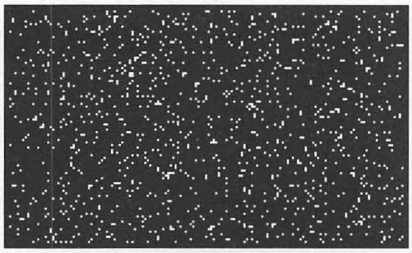
Dissolve is one of the nicest effects and does an excellent job of easing the transition between different cards. It is the effect that turns random pixels of one card to those of another card or image until all the pixels are changed. The entire process only takes an instant and has also been called *salt and peppering* or *random pixel-mixing.* Figure 3-3A shows the beginning of a dissolve. Figure 3-3B shows the same card just before it turns completely black. The next example provides a good contrast card for use in some of the later examples.

Visual Effects Need Browse Tool
`visual effect` only occurs when the browse tool is selected. If any other tool is selected, nothing happens.

Example 4:
1. Go to the Objects card in your HyperTalk Bible stack.



A. Dissolve to black 1.



B. Dissolve to black 2.

Figure 3-3. The dissolve effect.

2. Select New Card from the Edit menu. Choose the bucket tool, select a distinctive pattern from the Pattern palette (the pie lattice works well), and pour the pattern onto the card.
3. Go to the Audio-Visual card, make a new button, and name it `Dissolve`.
4. Replace the mouseUp handler in the Dissolve button with the following:

```
on mouseUp -- Dissolve button
  visual effect dissolve
  go prev
  visual effect dissolve
  go next
end mouseUp
```

5. Click the Dissolve button.

Result:

Your Audio-Visual card dissolves to the patterned card, which dissolves back to the Audio-Visual card.

Mac II and Color

There is a problem with doing visual effects on the Mac II. HyperCard's effects only show if the Characteristics in the Monitor section of the Control Panel is set to 2 colors or 2 grays.

Modifiers to Visual Effects

You can add to the visual impact of visual effects by controlling with speed modifiers how fast they execute. If you are using an accelerated Macintosh or a Mac II, you may want to stick with the slow or even the very slow modifier to compensate for the faster speed of your machine. Faster machines can perform visual effects so fast that you may miss them.

Example 5:

1. Go to the Audio-Visual card in your HyperTalk Bible stack.
2. Add a new card button and name it `Speed`.
3. Replace the mouseUp handler outline in the Speed button with the following:

```
on mouseUp -- Speed button
  visual effect dissolve very fast
  go prev
  visual effect dissolve very slowly
  go next
end mouseUp
```

4. Click the Speed button.

Result:

The Audio-Visual card appears to dissolve rapidly into the patterned card, which then dissolves slowly back to the Audio-Visual card.

You can combine visual effects and add an image modifier (see Figure 3-3, which shows a card dissolving to black rather than directly to the destination card). The image modifier allows you to fade to the destination card in the `go` command statement. You can fade to a black screen, to a gray screen, to a white screen, or to the inverse of the card (all black pixels turned to white and all white turned to black). The image modifier is especially valuable when moving between nearly identical cards. If you go directly from one card to another similar card, the user may not realize that a transition has occurred. To emphasize the transition, go to an intermediate contrasting image and *then* to the new card. The `to card` modifier is the default, so if you specify it or don't specify anything, the visual effect ends with the image of the card designated in the `go` command statement.

It is usually a good idea, when using the image modifier, to have one visual effect that ends with the intermediate image and a second effect that ends with the destination card. Dissolving from one card to a black, white, or gray screen and then dissolving again to the destination card can be an attractive effect.

Example 6:

1. Go to the Audio-Visual card in your HyperTalk Bible stack.
2. Add a new card button and name it `Double Dissolve`.
3. Replace the mouseUp handler in the Double Dissolve button with the following:

```
on mouseUp -- Double Dissolve button
  visual effect dissolve slow to gray
  visual effect dissolve slow
  go to card
end mouseUp
```

4. Click the Double Dissolve button.

Result:

First, the card dissolves slowly to a gray screen, and then the gray screen dissolves back to the original Audio-Visual card.

When a `visual effect` statement is executed, it is sent as a message to HyperCard. HyperCard holds onto it until a `go` message comes by or until the handler ends. As with the `play` command, there is a lapse of time between a statement's execution and the user sensing the effect. You might think that you could slow down

visual effects by placing `wait` commands between them, but this doesn't work. Keep in mind that any commands mixed with `visual effect` commands appear to be executed first.

Example 7:

1. Go to the Audio-Visual card in your HyperTalk Bible stack.
2. Add a new card button and name it `Dissolve To Nowhere`.
3. Replace the mouseUp handler outline in the Dissolve To Nowhere button with the following:

```
on mouseUp -- Dissolve To Nowhere button
  visual effect dissolve fast to black
  visual effect dissolve fast to card
  visual effect dissolve slow to white
  visual effect dissolve very slowly to inverse
  visual effect dissolve very fast to gray
  go to card
end mouseUp
```

4. Click the Dissolve To Nowhere button.

Result:

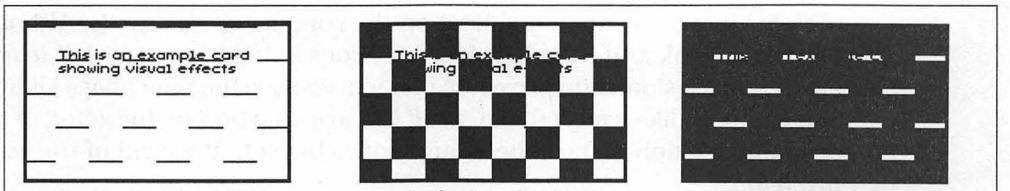
First, the card dissolves rapidly to a black screen. Next, the black screen dissolves back to the original Audio-Visual card. The card then dissolves slowly to a white screen, which dissolves even more slowly to the inverse of the Audio-Visual card. The inverted Audio-Visual card dissolves quickly to a gray screen before returning to its original appearance. Accomplishing each effect takes from a fraction of a second to several seconds, depending on the speed modifier.

In the last two examples, you used the `to card` option in the `go` command statement so that you could use visual effects without actually going anywhere. This shows that you can use the visual effects to provide visual feedback to actions that don't involve navigation. For example, for a dramatic effect, you might want a card to dissolve on itself when a user clicks a button.

Venetian Blinds and Checkerboards

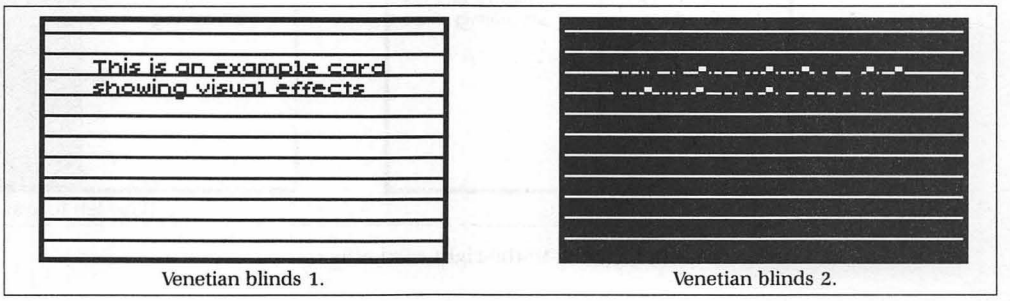
The dissolve option isn't the only visual effect. In the next example, you see some general purpose effects that have only one form, unlike the ones we see later that have different forms intended for particular purposes. The *checkerboard* effect divides the screen into squares, and then, starting at the top of each square, it fills with black until it reaches the bottom. Figure 3-4 shows three views of the checkerboard effect: A is the beginning, B is halfway through, and C shows it nearing the end.

The *venetian blinds* effect works the same way checkerboard does except that rather than squares, rectangles extend horizontally across the screen. Figure 3-5 shows the venetian blinds effect.



A. Checkerboard to black 1. B. Checkerboard to black 2. C. Checkerboard to black 3.

Figure 3-4. The checkerboard effect.



Venetian blinds 1. Venetian blinds 2.

Figure 3-5. The venetian blinds effect.

Example 8:

1. Go to the Audio-Visual card in your HyperTalk Bible stack.
2. Add a new card button and name it `Fancy`.
3. Replace the mouseUp handler in the Fancy button with the following:

```
on mouseUp -- Fancy button
  visual effect venetian blinds slowly to black
  visual effect checkerboard slowly to white
  visual effect dissolve slowly to card
  go to card
end mouseUp
```

4. Click the Fancy button.

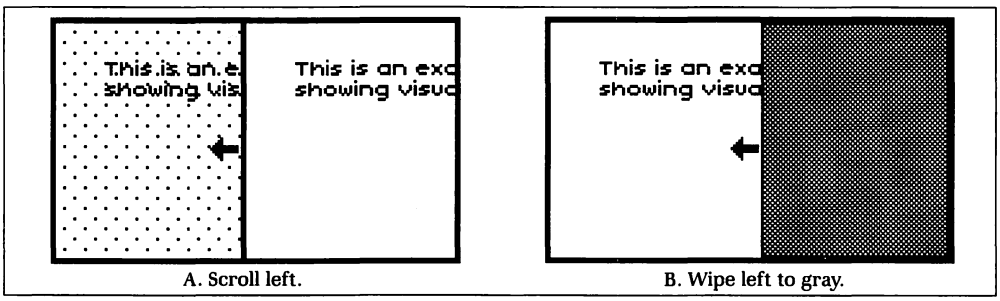
Result:

Horizontal black lines expand until the entire screen is black, white squares form, turning the entire screen to white, and finally, the Audio-Visual card dissolves into view.

Movie Special Effects 1: Scroll and Wipe

The remaining effects require a little thought to ensure their proper use. They form a collection that gives movie-like lens controls—effects like those you have seen in Star Wars movies. The *wipe...* effect, a favorite of George Lucas, causes a vertical or horizontal line to pass over the screen with the new card visible behind the advancing line. It usually means "meanwhile, back at the ranch." All visual effects are

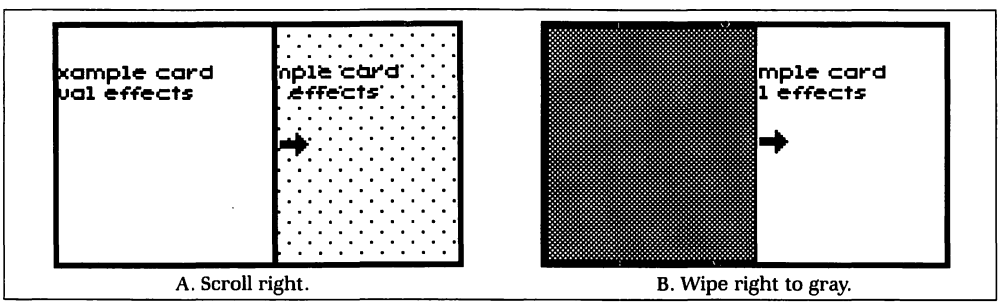
merely visual; they have no effect on the contents of the cards. Although the wipe effect may look as if it is changing the values in fields on a card, it is not. The *scroll* effect is like a slow slide projector that lets you see the new image sliding into place. Some effects, like *wipe left* and *scroll left,* are appropriate for going to the next card or to information that can be thought of as being to the right of the screen (see Figure 3-6).



A. Scroll left. B. Wipe left to gray.

Figure 3-6. Going-to-the-right card effects.

Others, like *wipe right* and *scroll right,* should be used when you go to the previous card or to information off to the left (see Figure 3-7).



A. Scroll right. B. Wipe right to gray.

Figure 3-7. Going-to-the-left card effects.

To make it appear that the information came down from the top of the screen, use *wipe down* or *scroll down* (see Figure 3-8), and to make it appear that information came up from the bottom of the screen, use *wipe up* or *scroll up* (see Figure 3-9).

For information, especially graphics information, that won't fit on a single card, you can organize cards in your stack into a larger virtual card and use control buttons to scroll around in the larger space (see Figure 3-10).

Example 9:

1. Go to the Audio-Visual card in your HyperTalk Bible stack.
2. Add a new card button and name it `Pan`.
3. Replace the mouseUp handler in the Pan button with the following:

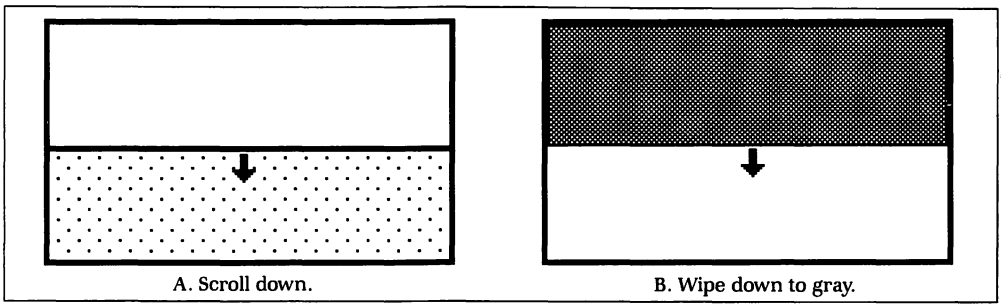


Figure 3-8. Going-to-the-card above effects.

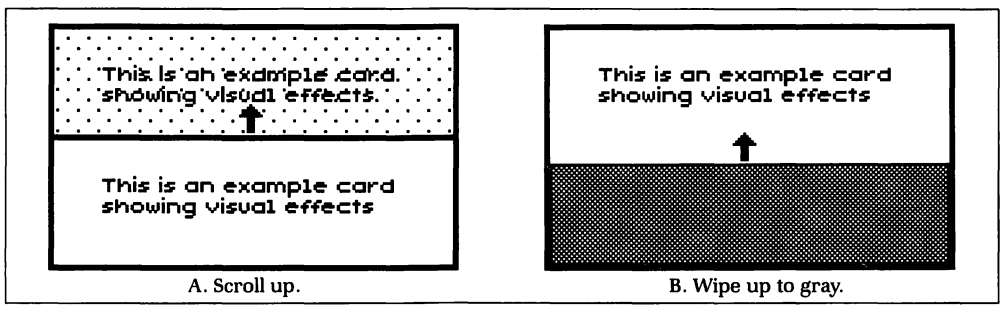


Figure 3-9. Going-to-the-card below effects.

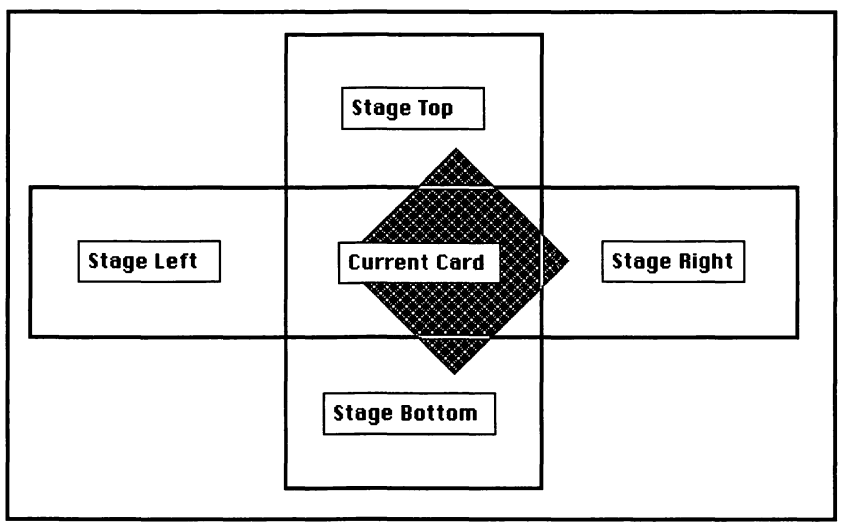


Figure 3-10. A special effects view of the larger virtual graphic area.

```
on mouseUp -- Pan button
  visual effect scroll right slow
  visual effect scroll down slow
  visual effect scroll left slow
  go to previous
  visual effect scroll up slow
  go to next
end mouseUp
```

4. Click the Pan button.

Result:

The patterned card appears to slide in from the left, and then it slides down from above, slides in from the right, and finally it slides up from below.

The next example shows two versions of an interesting wipe effect that mimics slides being loaded in a projector with light showing between slides. You can use either the horizontal or vertical version in your own scripts.

Example 10:

1. Go to the Audio-Visual card in your HyperTalk Bible stack.
2. Add a new card button and name it `Double Wipe`.
3. Replace the mouseUp handler in the Double Wipe button with the following:

```
on mouseUp -- Double Wipe button
  -- horizontal double wipe
  visual effect wipe right slowly to white
  visual effect wipe left slowly to card
  go to next
  -- vertical double wipe
  visual effect wipe up slowly to white
  visual effect wipe down slowly to card
  go to prev
end mouseUp
```

4. Click the Double Wipe button.

Result:

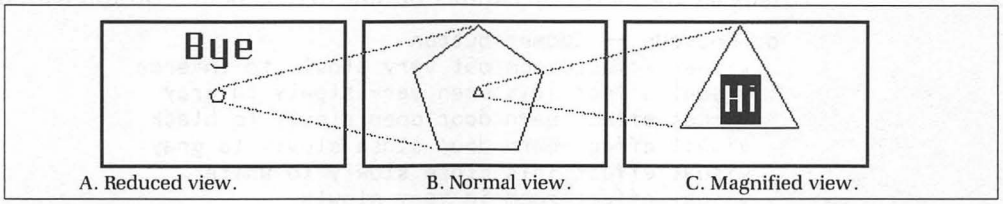
A wave of white washes over the screen from left to right and then rebounds back to the left to show the patterned card. Another wave of white washes up from the bottom of the screen to the top and rebounds down to show the Audio-Visual card once again.

Feel free to try other combinations of these effects with different images and at different speeds. Hundreds of possible combinations are possible in a single statement and thousands of possible combinations of statements are available.

Movie Special Effects 2: Zoom, Scroll, and Barn Doors

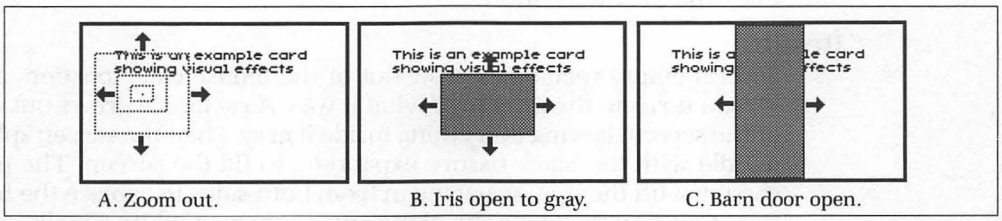
Sometimes you want to give the impression that you are zooming in for a more detailed view of a graphics image or that you are zooming out for a more comprehen-

sive view of a subject. Given the situation shown in Figure 3-11, you could use *iris out* when moving from the left to the right, which gives the impression of penetrating into the graphic, or you could use *iris in* when moving from right to left to simulate pulling back from the images. Figure 3-12 shows how you can use zoom out, iris out, or even *barn door open* for penetrating effects.



A. Reduced view. B. Normal view. C. Magnified view.

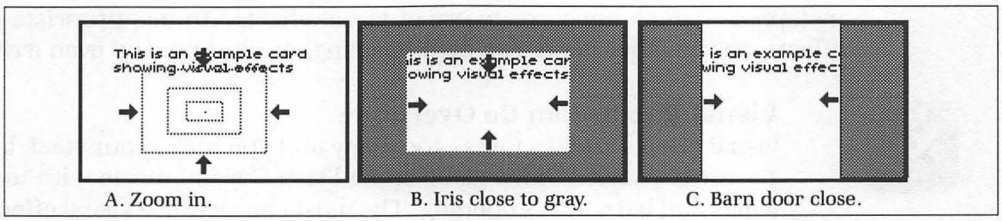
Figure 3-11. View of cards at different detail levels.



A. Zoom out. B. Iris open to gray. C. Barn door open.

Figure 3-12. Going-in effects.

When you are going back to a less detailed view, use *zoom in,* iris in, or *barn door close* as shown in Figure 3-13.



A. Zoom in. B. Iris close to gray. C. Barn door close.

Figure 3-13. Going-out effects.

Barn door is like a double wipe with two vertical lines wiping out from the center on barn door open or wiping in to the center on barn door close. Barn door effects can be used to signal the unveiling of new information. The iris effect is actually a square that grows out of a point at the center of the screen in the case of iris open, or shrinks to that point in the case of iris close. The *zoom open* effect causes several rectangles to zoom out from the current mouse cursor coordinates. The zoom open effect is the one you get when you double click icons on the Macintosh

desktop. As the next example shows, there is really no limit to the kind and number of effects you can produce using the `visual effect`command.

Example 11:

1. Go to the Audio-Visual card in your HyperTalk Bible stack.
2. Add a new card button and name it `Zoomer`.
3. Replace the mouseUp handler outline in the Zoomer button with the following:

```
on mouseUp -- Zoomer button
  visual effect zoom out very slowly to inverse
  visual effect iris open very slowly to gray
  visual effect barn door open slowly to black
  visual effect barn door close slowly to gray
  visual effect iris close slowly to white
  visual effect zoom in very slowly
  go to card
end mouseUp
```

4. Click the Zoomer button.

Result:

Several empty rectangles grow out of the mouse click location, and then the screen turns to the inverse of what it was. A rectangle grows out of the center of the screen, leaving everything inside it gray. Then the screen splits down the middle with the black fissure expanding to fill the screen. The process is reversed, with the gray sweeping in from both sides to replace the black and the resulting gray rectangle shrinking into a screen of white. Finally, several empty rectangles shrink to the center of the screen and the card is revealed again.

Although visual effects may seem a bit frivolous, they are handy for providing visual feedback on how you are moving through the data in your stack. It often is hard to work with information in a computer because it is hard to conceptualize its structure. HyperCard's stack metaphor helps the user, and your scripts can reinforce the metaphor with an appropriate use of visual effects. An inappropriate use of visual effects, on the other hand, can be distracting, misleading, and even irritating.

Visual Effects Can Be Overdone

Visual effects are like fonts—too many and you make your stack look like a cross between a video game and a Flash Gordon movie with too many wipes and barn doors opening. The most conservative visual effect is dissolve, which you can use with the fast modifiers to make slick transitions from card to card.

The *doMenu* Command

Frequently, you need to perform the same actions in your scripts that you normally perform by selecting items from the menus. You may want to make a new stack or a new button, compact the stack, or perform one of the Paint menu actions. Instead of

having HyperTalk commands that duplicate the HyperCard menu items, HyperTalk gives you one command, `doMenu`, that can do everything the menus can do. (See Figure 3-14 for the syntax of this command.) Actually, the `doMenu` command simply lets you access all of HyperCard's menu items by passing the item name as a parameter of the command. We say more about parameters later, but for now, think of them as information sent along with the command. For example, in the statement `beep 3`, `3` is the information passed as a parameter that tells HyperCard how many times to beep.

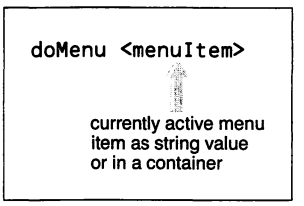


Figure 3-14. The `doMenu` command syntax.

We preview this command now because you need to use it from time to time before you read chapter 16, "Simulation," where it is covered in detail. The `menuItem` parameter has to be an exact match of the HyperCard menu item—and that item must be active—for the `doMenu` command to work. For instance, to open the Stack Info dialog box from a script, use the command `doMenu "Stack Info..."`, where the three dots are required and are periods, not an ellipsis. In the next example, you make a button that compacts your HyperTalk Bible stack to remove any free space that has built up.

Example 12:

1. Go to the Audio-Visual card in your HyperTalk Bible stack.
2. Add a new card button and name it `Compact`.
3. Replace the mouseUp handler in the Compact button with the following:

```
on mouseUp -- Compact button
  doMenu "Compact Stack"
end mouseUp
```

4. Click the Compact button.

Result:

The beach ball cursor is displayed while the stack is compacted to remove all free space.

You can even open desk accessories with the statement `doMenu "Control Panel"`. The `doMenu` command can be used to turn any menu operation into a button operation. Consider the task of creating a new button and then naming that button, a task that has been repeated in almost every example in this chapter. This normally requires that you pull down the Objects menu twice and make two menu item selections. First, you must select the New Button item, and then you must select the Button Info item (or else double-click the button). If you select the wrong item by

mistake (these things do happen), you may have to do even more work. As an example of how you can use the doMenu command to your advantage, the next example provides a background button that performs both tasks with much less chance of error.

Example 13:

1. Go to the Audio-Visual card in your HyperTalk Bible stack.
2. Select Background in the Edit menu and then New Button from the Objects menu.
3. Select Button Info... from the Objects menu and name the new background button Add a Button.
4. Replace the mouseUp handler in the Add a Button button with the following:

```
on mouseUp -- Add a Button button
  doMenu "New Button"
  doMenu "Button Info..."
end mouseUp
```

5. Click the Add a Button button.

Result:

A button is created and the Button Info dialog box is opened. Type Button Tool into the name field.

The *choose* Command

The doMenu command can't be used to select tools from the Tool Palette, but the choose command can. Figure 3-15 shows the syntax for this command.

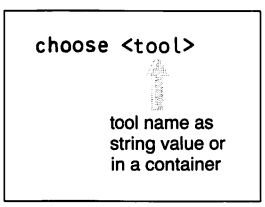
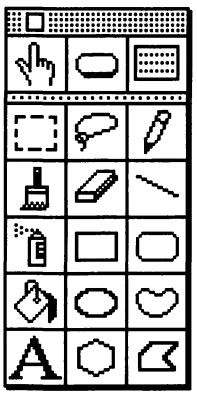


Figure 3-15. The choose command syntax.

The choose command is covered more extensively in chapter 16, "Simulation" and in chapter 17, "Graphics ", but this preview should tell you enough about it so that you can use it in the intervening chapters. In Figure 3-16, we even show you the full list of tool options in case you want to experiment on your own.

In the last example, we created a button named Button Tool, which we put a handler into in the next example. Let's say the purpose of this button is to permit you to choose the button tool without having to pull out the Tool Palette.



Legend:		
browse tool	button tool	field tool
select tool	lasso tool	pencil tool
brush tool	eraser tool	line tool
spray tool	rectangle tool	round rect tool
bucket tool	oval tool	curve tool
text tool	regular polygon tool	polygon tool

Figure 3-16. Tool names.

Example 14:

1. Go to the Audio-Visual card in your HyperTalk Bible stack.
2. Open the Script Editor for the Button Tool button and replace its mouseUp handler with the following:

```
on mouseUp -- Button Tool button
  choose button tool
end mouseUp
```

3. Click the Button Tool button.

Result:

All buttons on the screen are outlined to show that the button tool has been selected.

If you think the Button Tool button will save time when you need to switch tools to edit a button, you should move it into the background, where the Add a Button button already is. This might be a good place to remind you that only the browse tool can send messages to a button. You can write button handlers using the `choose` command to switch to any tool from the browse tool, but once another tool is chosen, you can't use a button to switch tools again.

What you can do with the `choose` command is pretty limited until we introduce some new commands in chapter 16, "Simulations." Still, with the `doMenu` and `choose` commands, you now have the ability to select all the menu items and all the tools under script control, which is an important first step on the road to full HyperCard simulation from within your HyperTalk scripts.

The *answer* Command

As a Macintosh user, you are familiar with modal dialog boxes—the little windows that make sure you know about something (and force you to click an OK button to make them go away) or that let you make a decision by clicking a button. Alert boxes are a subset of the Mac's modal dialog boxes and warn the user of the dangers ahead. All modal dialog boxes prevent the user from using the menu bar or anything else on the screen until the box itself is dealt with.

We'll be using modal dialog boxes throughout the book, so let's learn how they work. The HyperTalk command `answer` lets you display a simple modal dialog box from a script. We refer to this simple box as an *answer box*. You determine the prompt text displayed in the answer box and what the text in the reply buttons will be. Unfortunately, you have no control over the placement of the answer box on the screen. On a small screen Mac, it always is positioned as you see it in Figure 3-17.

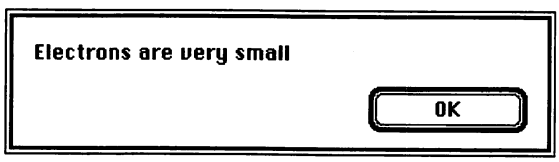


Figure 3-17. Macintosh answer box.

On a larger screen, it is positioned at the same distance from the top of the screen and centered horizontally but the HyperCard window is centered both horizontally and vertically, so the answer box may not be over the HyperCard window. Figure 3-18 shows the `answer` command syntax.

```
answer <question> [with <reply> [or <reply> [or  <reply>]]]
```

prompt string as string value or in a container

button string as string value or in a container

Figure 3-18. The `answer` command syntax.

An answer box can be set up to have one, two, or three reply buttons. If you don't provide any reply options (the text that goes in the button), the default version of the answer box is displayed with just an OK button that can be selected by clicking in the button or by pressing the Return key. If you do provide reply options, the last (farthest to the right in the command statement) option is selectable by pressing the Return key. The last button also has a double border to indicate its special (default) status. The number of characters displayed in the prompt string of the answer box

varies because a proportional font, Chicago, is used. The number of prompt string characters is approximately 46. The maximum number of characters in a reply button is approximately 11. (Reply text is centered, so if your text is wider than the button, you lose both the beginning and the end.)

Example 15:

1. Go to the Audio-Visual card in your HyperTalk Bible stack.
2. Add a new card button and name it `Answer`.
3. Replace the mouseUp handler in the Answer button with the following:

```
on mouseUp -- Answer button
  answer "Electrons are very small"
  visual effect dissolve
  go prev
  wait 30 ticks
  go next
end mouseUp
```

4. Choose the browse tool and then click the Answer button.

Result:

An answer box appears with the text `Electrons are very small` and a default OK button (see Figure 3-17). Click Ok to get rid of the box and to trigger the visual effect. The dissolve ends by showing the previous card in the stack. After a half-second pause, the Audio-Visual card reappears.

In this example, program flow halts twice: once at the `answer` command (until the user clicks a button) and again at the `wait` command for 30 ticks or half a second. Because this is your first look at the `wait` command, we should probably pause here to explain how it works.

The *wait* Command

There are actually a number of versions of the `wait` command, but for now, we just look at the simplest one with the syntax shown in Figure 3-19.

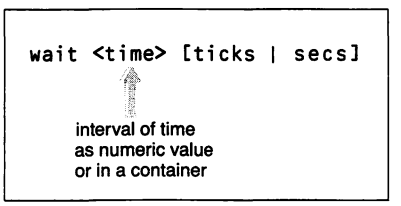


Figure 3-19. The `wait` command syntax.

This command causes program flow to halt until the time you load in the timer runs out. You measure the time in either ticks or seconds, but if you don't specify

seconds (or secs), HyperCard assumes you mean ticks. What is a tick? As you may have guessed from our statement that the 30 ticks in the last example was half a second, a *tick* is a sixtieth of a second. Consequently, each of the following `wait` statements have the same meaning

```
wait 120
wait 120 ticks
wait 2 seconds
wait 2 secs
```

The other versions of the `wait` command are covered in chapter 10, "Repeats."

Copying a Card Copies Its Background
Be careful when copying cards from one stack to another. When you copy a card, you also copy its background. When a copied card is pasted into another stack, the background is pasted, too. The only way around this is to copy the individual objects of the card: graphics, buttons, and fields. This can be laborious on cards with many buttons and fields. The public domain script Groupies is a must for moving multiple objects. It's available on AppleLink.

The *answer* Command Revisited

The next example shows an answer box that is used to give the user several options. The content of the selected button is put into the global variable `it` (more about `it` in the next chapter).

Example 16:

1. Go to the Audio-Visual card in your HyperTalk Bible stack.
2. Add a new card button and name it `Card-Navigator`.
3. Add a new card after the Audio-Visual card and paint `Next` on it with one of the paint tools.
4. Add another card and label it `Last`.
5. Replace the mouseUp handler in the Card-Navigator button with the following:

```
on mouseUp -- Card-Navigator button
  answer "Go to which card?" with "First" or "Last"
or "Next"
  play "harpsichord" tempo 15 "32"
  visual effect dissolve slow
  doMenu it
end mouseUp
```

6. Click the Card-Navigator button.
7. Press the Return Key.

Result:

The answer box disappears and a low sound plays as the card dissolves slowly into the Next card.

8. Return to the Audio-Visual card (Command-~), click the Card-Navigator button, and then click the Last button.

Result:

The answer box disappears and a low sound plays as the card dissolves slowly into the Last card.

9. Return to the Audio-Visual card (Command-~), click the `Navigator` button, and then click the `First` button.

Result:

The answer box disappears and low sound plays as the card dissolves slowly into the first card in the stack.

There are a number of things you can do with the `answer` command based on the particular button pressed, but we can't use those button results until we have learned about branching in chapter 9, "Ifs." However, even now, you can use `answer` to show the results of statement execution in your scripts and to communicate with the user. The `answer` command is also, as you will see, invaluable when you are debugging scripts, because it lets you pause in the middle of a handler and examine the current status of containers (more on containers in the next chapter).

The *ask* Command

Sometimes it isn't feasible to provide the user with only three 11-character-per-label button choices. Sometimes you need to give the user the chance to actually type a long answer. The Macintosh way to get this kind of information from a user is with another type of modal dialog box (see Figure 3-20). This dialog box provides a field that the user can type in along with buttons to signal when the user is through with the box. The `ask` command displays a modal dialog box with a script-defined prompt string across the top, a text field in the middle (which may hold a script-defined default string), and a pair of buttons at the bottom. Figure 3-21 shows the `ask` command syntax.

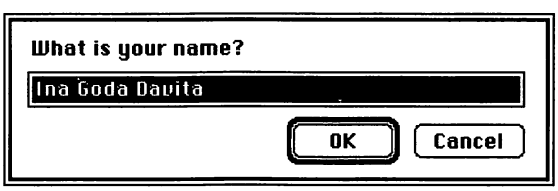


Figure 3-20. The `ask` type modal dialog box.

As with the `answer` command, program flow stops at the `ask` command state-

ment until the user selects a button or presses the Return key. The user's response is again placed in the global variable `it`. If the Cancel button is clicked, the HyperTalk constant `empty` is put into `it`. `it` and `empty` are covered in the next chapter, but for now, think of `it` as a holding tank for results and `empty` as indicating that nothing else is in the holding tank. You can try clicking the Cancel button in the next example to see what happens.

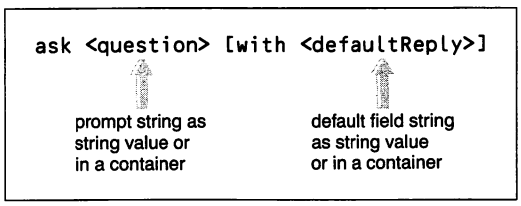


Figure 3-21. The `ask` command syntax.

Example 17:

1. Go to the Audio-Visual card in your HyperTalk Bible stack.
2. Add a new button and name it `Ask`.
3. Replace the mouseUp handler in the Ask button with the following:

```
on mouseUp -- Ask button
  ask "What is your name?" with "Ina Goda Davita"
  put it into message box
end mouseUp
```

4. Click the Ask button.

Result:

A modal dialog box is displayed with the question `What is your name?` at the top, the prompt string `Ina Goda Davita` selected in the middle, and OK and Cancel buttons at the bottom.

5. Click the OK button.

Result:

The modal dialog box disappears and `Ina Goda Davita` appears in the Message box.

6. Click the Ask button again. This time, type `Born Toby Wild` in the dialog box field.
7. Click the OK button.

Result:

The modal dialog box disappears, and `Born Toby Wild` appears in the Message box.

The `ask` command is an essential part of the Macintosh user interface. It also is the easiest way to get detailed information from the user into your scripts. Normally, `answer` presents simple choices, whereas `ask` causes your scripts to look for more detailed information. The script often has to analyze the string that the user enters. After you read chapter 4, "Containers and Values" and chapter 9, "Ifs," you will find that the `ask` command (and the `answer` command) are even more valuable.

Extending *ask* and *answer*

If you are used to languages with better access to Macintosh resources, you may find the `ask` and `answer` commands limiting. The down side to their remarkable simplicity is that you lack real control over the actual resource components. If you need more control, you might look into the `DoList` XFCN, which can be used with `ResEdit` to greatly extend your dialog box options. `DoList` is an XFNC that shows the user a standard list type dialog box like the one in open dialog boxes. In addition to this standard function, `DoList` also lets you pass information to other dialog boxes, which you can construct using `ResEdit`, and then returns the user's side of the dialog. For more information on using XFNCs and XCMDs, see Chapter 20, "XCMDS ans XFCNS."

Summary

In this chapter, you learned how to use the `play` command, change the voice, vary the tempo, and control the actual notes played. You saw how the `visual effect` command can be used to enhance card transitions. You sampled a variety of effects and experimented with the speed and image modifiers.

You saw how you can use the `doMenu` and `choose` commands to simulate the selecting of menu items and Tool Palette tools. Finally, you learned how to communicate or interact with the user of a stack using the `answer` and `ask` commands to write your own modal dialog boxes. You also had a quick look at the `wait` command.

We only scratched the surface of what these commands can do, but the fact that we can even introduce such powerful commands so early in the book is a tribute to the HyperTalk language and the people who created it. As with HyperCard, the goal of HyperTalk is to provide a software tool that is accessible to the novice, yet flexible and powerful enough for the experienced user.

In the next chapter, we introduce you to HyperTalk containers, including variables—possibly the most powerful feature of any programming language.

Review Questions

1. What `play` command parameter would you change to make notes play faster?
2. What `play` command voices are included with HyperCard?
3. What are the two ways to designate notes for the `play` command?
4. List the three errors in the following statement.

```
play "a b h" tempo 120
```

5. Write a statement to play the notes F and G with a tempo of 150 using the harpsichord voice.

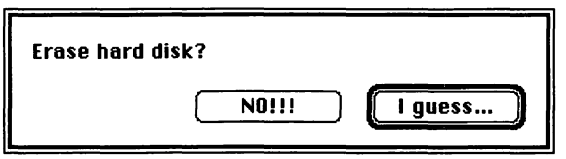
6. What is wrong with the statement

```
visual dissolve
```

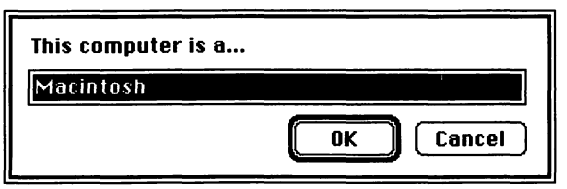
7. List the four errors in the following statement.

```
visual effect fast venetian blind close to next card
```

8. Which command would you use to switch to field editing mode?
9. Which command would you use to open the background dialog box?
10. Name two HyperTalk commands that stop program flow until the user clicks a button.
11. What would the statement to create the dialog box shown below look like?



12. What would the statement to create the dialog box shown below look like?

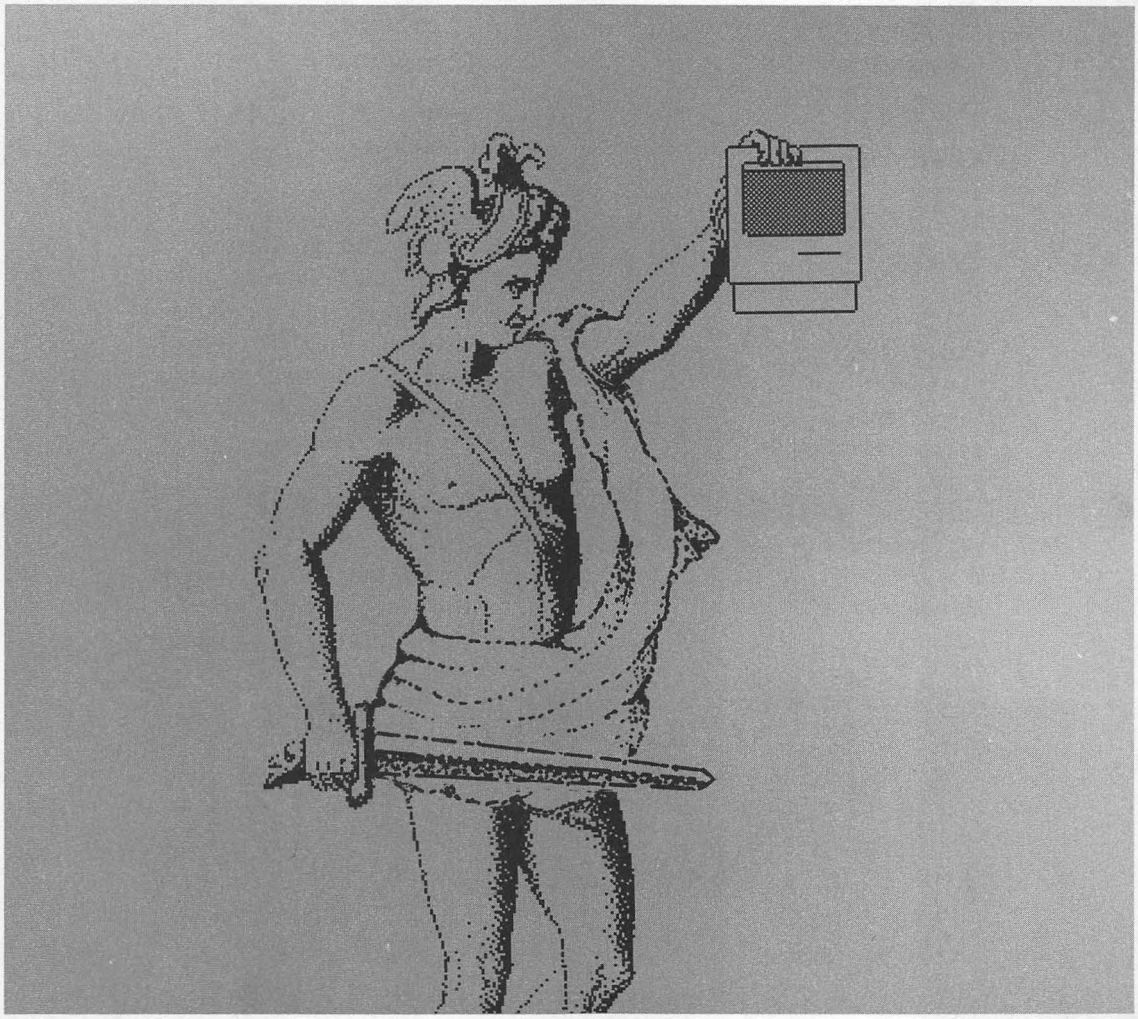


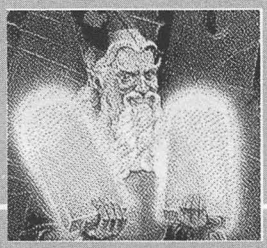
Exercises

1. Write a handler that lets the user choose a voice, and then play several notes in that voice.
2. Write a handler that triggers a checkerboard visual effect to the inverse of the Audio-Visual card, followed by a gradual dissolve back to the card, each time the mouse button is pressed in the card.
3. Write a handler for one button that lets the user choose to select the button tool or the field tool. Also, provide audio feedback in the form of a fast middle-C boing.

Chapter 4

Containers and Values





Quick View

Commands
`get`

Keywords
`global`

Variables
`it`

Functions
`the number`
`the selection`
`selectedFields`
`selectedLine`
`selectedChunk`
`selectedText`

Properties
`name`

Concepts
Values
Containers
Variables
Sources for values
Strings and numbers
HyperTalk constants
Functions and Properties
Fields
Text and number the same
Variables are containers
Good and bad names
Global variables
The `idle` message
Local variables
`it` and `get`
`the selection`
Message box as a container

Chapter 4

Containers and Values

In this chapter, you are going to learn about the values and containers that HyperTalk provides for storing information. HyperTalk offers a plentiful assortment of containers that are easier to use than the variables of other languages. (Think of them as smart variables.) Besides variables, HyperTalk offers the scriptor fields, `the selection`, and the Message box as storage places for values.

Before we go any further, however, let's answer a few questions that are probably popping into your head if you are new to programming. These questions are, "What is a value? What are variables and containers?"

What's a Value?

Let's start with values. A *value* is anything that you can type into a field, and it's other things, too. A value can be a number, a letter, a word, a date, a sentence, or several pages of text. If you think of HyperTalk and HyperCard as a machine, values are the materials worked, or processed, by the machine.

What Are Containers and Variables?

To prevent our values from "leaking all over the machine," we have to keep them in containers. The traditional term for value containers is *variable*. A variable is a kind of container that does not show up on the screen and is only represented in a script by a letter or a word. A variable is a chameleon-like entity that has a value which can be changed at any time. The variable `var1` can contain Hello one minute and Goodbye the next.

Example 1:

1. Open your Home stack, show the Message box, enter the following line in it, and press the Return key:

```
put "Hello" into var1
```

2. Enter the following line in the Message box and press the Return key:

```
var1
```

Result:

`Hello` appears in the Message box. See Figure 4-1.

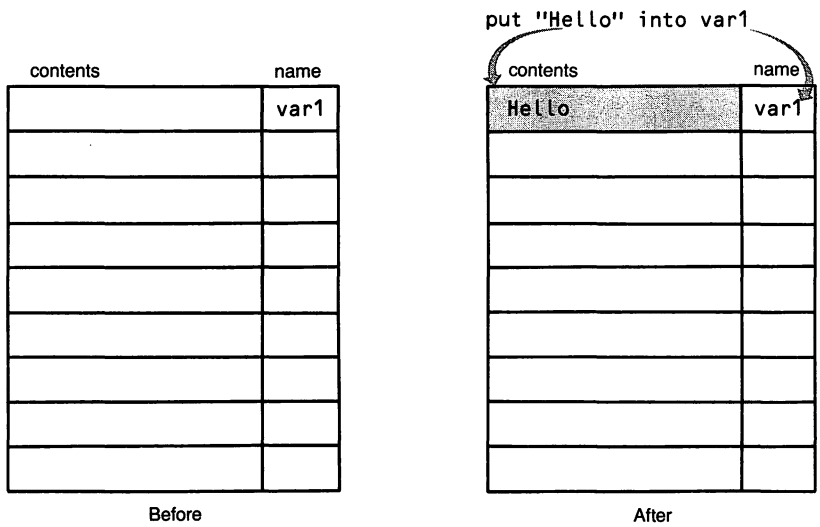


Figure 4-1. Assigning a value to a variable.

In Figure 4-1 (and also in Figures 4-2 through 4-5 and Figure 4-7), the statement that assigns a value to a variable is shown above the two boxes. The left box shows the value of the variable named prior to the assignment, and the right box shows the value after the assignment. In Figure 4-1, for example, the variable `var1` starts with no value (it doesn't exist) and then has a value of Hello after the assignment.

3. Enter the following line in the Message box:

```
put "Goodbye" into var1
```

4. Enter the following line in the Message box:

```
var1
```

Result:

`Goodbye` appears in the Message box. See Figure 4-2.

Figure 4-2 shows that the variable `var1` started containing the value Hello but that the value was changed to Goodbye by the assignment statement `put "Goodbye" into var1`. The beauty of the variable's "variable" nature is that you can use the same statement, `put var1 into the message box`, for instance, to say both Hello and Goodbye. This capability of a variable name to stand for many different values is one of the foundation stones of programming. The capability of HyperTalk vari-

ables to stand for any kind of value (numeric, text, date, time) is one of the most powerful features of HyperTalk.

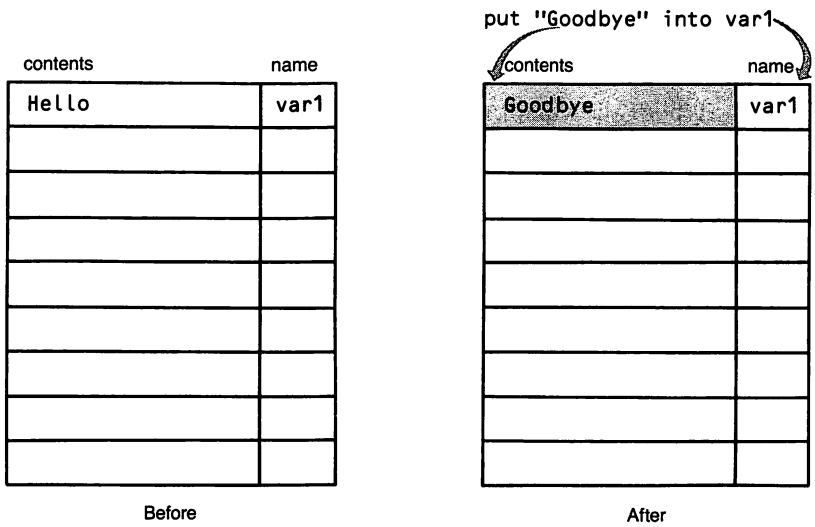


Figure 4-2. Reassigning a value to a variable.

The importance of variables in programming cannot be overstated. If you've got the idea already, you can skip the rest of the next section, but if you have any doubts, keep reading.

More About Variables

A variable is like a cubbyhole in a rolltop desk where values can be stored temporarily. Imagine a neatly organized desk with a row of these cubbyholes labeled CubbyHole1 through CubbyHole5. If you always put tax receipts into CubbyHole2 and unpaid bills into CubbyHole4, you could use the term *CubbyHole2* to mean the receipts in that container. You might say to an assistant, "Get me CubbyHole2" and expect to receive a bunch of receipts, not the cubbyhole itself ripped out of your desk. If you came into some money and decided to clear out your unpaid bills and use that space for I.O.U. notices from your needy friends, you would first ask your assistant to get you CubbyHole4 so that you could write checks for all the bills there, and then tell your assistant to put the I.O.U. notes into CubbyHole4.

When you program a computer, the cubbyholes are variables (or any containers in HyperTalk), and your assistant is the computer itself. You could name a variable `CubbyHole2` and use the command `put "HyperCard, $50.00, 8/20/87, Mac's R'us" into CubbyHole2` to put this receipt information into the variable. The biggest

difference between variables and cubbyholes is that variables exist in RAM and disappear when the computer is shut off.

Sources of Values

In HyperTalk, values can come from many sources. These sources include quoted and unquoted strings, numbers, constants, functions, properties, and containers. Hello and Goodbye, in the last example, were instances of quoted strings used as sources of value. In the next several sections, we demonstrate the other sources of value and tell you more about values.

Strings

Strings consist of groups of characters like `"This is a string"`, `"1,233.45"`, or any other combination of characters except double-quotation marks, return characters, and option-return characters. Strings should be set off by quotation marks, but there are cases (explained after Example 2) when quotation marks are optional.

Example 2:

1. Enter the following line in the Message box:

   ```
   put HyperCard into var2
   ```

2. Enter the following line in the Message box and press the Return key:

   ```
   var2
   ```

Result:

`hypercard` appears in the Message box. See Figure 4-3.

In Figure 4-3, the successful placing of the value `HyperCard` into the new variable `var2` is confirmed by the box on the right. Note that the uppercase letters in the unquoted string are converted to lowercase letters in the Message box. Because there are times when HyperCard can confuse unquoted strings with variable names, we recommend that you always try to put strings between quotation marks. Although one-word strings that don't duplicate variable names can be interpreted without quotation marks, if the string contains a space, it must be quoted. Strings are covered in greater detail in chapter 11, "Strings."

Numbers

HyperCard understands two kinds of number values: all numeric integers like 1234, and real numbers like 123.4 (with one decimal point and no commas). Numbers need not be quoted.

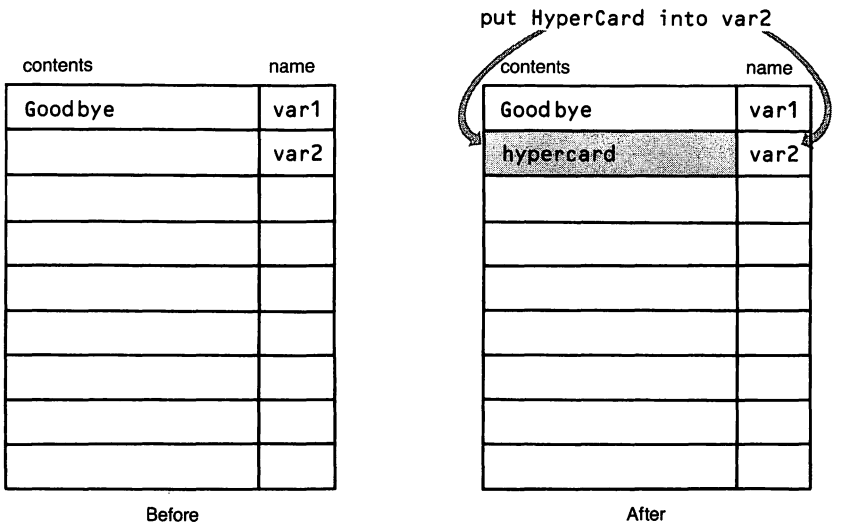


Figure 4-3. Assigning an unquoted string value to a variable.

Example 3:

1. Enter the following line in the Message box and press the Return key:

```
put 22 into var3
```

2. Enter the following line in the Message box and press the Return key:

```
var3
```

Result:

22 appears in the Message box.

3. Enter the following lines in the Message box and press the Return key after each line:

```
put 44 into var4
var4
```

Result:

44 appears in the Message box. See Figure 4-4.

Even when a value starts as a number, if it is stored in a container, it is converted to a character string. This is not a problem because HyperTalk knows to convert it back to a number for mathematic operations.

Secrets of the Message Box

If you make a mistake in the Message box, you move the insertion cursor to the mistake and correct it. Then, you can press the Return key even though the cursor is not at the end of the line. HyperTalk executes what is in the Message box.

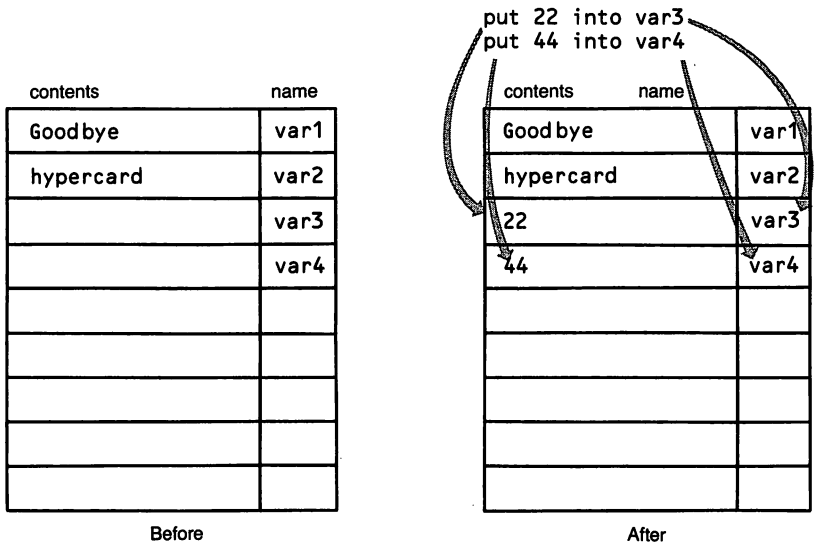


Figure 4-4. Assigning numeric values to variables.

Example 4:

1. Type the following lines into the Message box and press the Return key:

   ```
   put var3 + var4 into var5
   ```

2. Enter the following line in the Message box and press the Return key:

   ```
   var5
   ```

Result:

66 appears in the Message box. See Figure 4-5.

Figure 4-5 shows that the string values in `var3` and `var4` can be converted to numbers when used with the + math operator. Then, the result of this operation is put into the new variable `var5`. Not even HyperTalk can add a number to letters, as you can prove by trying to add `var3` to `var2` (which now contains the word `hypercard`).

Constants

You have already seen some of HyperTalk's *constants*. You can find some of the constants commonly used as sources of value in the following list.

Constants Used as Sources of Value

Numeric Constants:

zero, one, two, three, four, five, six, seven, eight, nine, ten

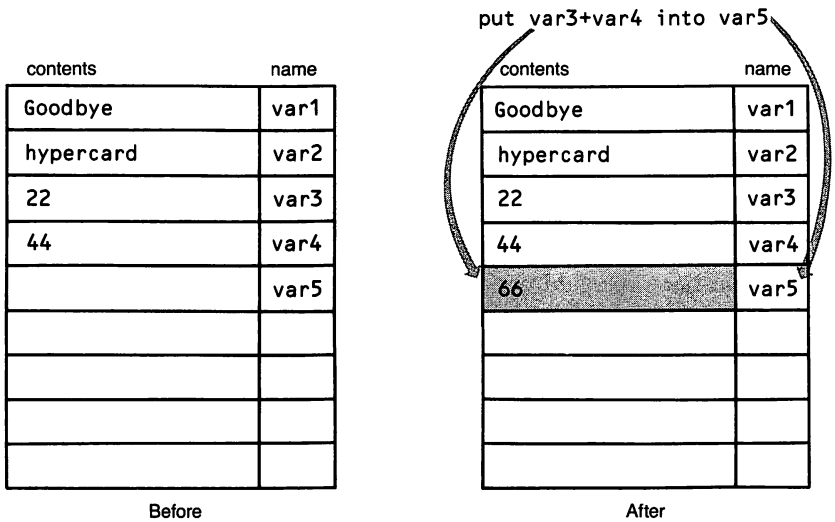


Figure 4-5. Converting strings to numbers and assigning the resulting value to a variable.

Other Character Constants:
return, tab, space, empty, quote

Miscellaneous Constants:
up, down, true, false

Constants are like strings in that they always have the same value—hence the term constant—but they are like variables in that the value is not the same thing as the characters that comprise the constant. The constant `one`, for example, has the same value as the number `1`, but does not have the same value as the string `"one"`. The constant `return` stands for the return character, which is generated when you press the Return key (ASCII 13).

Example 5:

1. Enter the following lines in the Message box and press the Return key:

   ```
   put six into var6
   ```

2. Enter the following line in the Message box and press the Return key:

   ```
   var6
   ```

Result:

6 appears in the Message box.

One constant frequently used with containers, such as fields, is `empty`. The empty character (ASCII 0) can be put into a container, or into a chunk of a container,

to displace that container's previous value. The command `put empty into var2` has the effect of dumping whatever was in `var2` and replacing it with the empty character. If followed by the command `put var2 into the Message box`, it would display `empty` in the Message box. As you recall from the last chapter, when an answer or ask box is canceled, `empty` is put into the variable `it` to let you know that there is no significant dialog.

Functions and Properties

You have worked with *functions* and properties in earlier chapters. As sources of values, there is little difference between functions and properties. Each can return strings, numbers, or constants, depending on the particular function or property in question. Some examples of functions are `the target`, which returns the name of an object, `random()`, which returns a number, and `the number of`.

the number of Function

Not to be confused with the relational number property of objects, `the number of` function returns the total number of objects of a specified type. The syntax for `the number of` is shown in Figure 4-6.

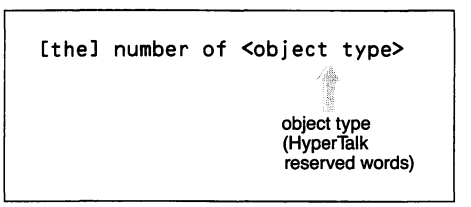


Figure 4-6. The `the number of` function syntax.

You can use this function to find the number of cards in a stack, the number of fields (or buttons) that belong to a card, or the number of fields (or buttons) that belong to the current background. An example of `the number of` function is `the number of cards`, which returns a number value, as demonstrated in the next example.

Example 6:

1. Enter the following lines in the Message box and press the Return key:

   ```
   put the number of cards into var7
   ```

2. Enter the following line in the Message box and press the Return key:

   ```
   var7
   ```

Result:

5 appears in the Message box. (The number depends on the number of cards in your Home stack).

Functions are covered in more detail in chapter 5, "Simple Math."

Properties

We have already introduced quite a few properties—from the global property `the userLevel` to the object properties `the number`, `the id`, `the name`, `the rect`, etc. An example of a property is `the name of this card`, which returns an object type and a quoted string (presuming the card is named).

Example 7:

1. Enter the following lines in the Message box and press the Return key:

   ```
   put the name of this card into var8
   ```

2. Enter the following line in the Message box and press the Return key:

   ```
   var8
   ```

Result:

`card "Home"` appears in the Message box. See Figure 4-7.

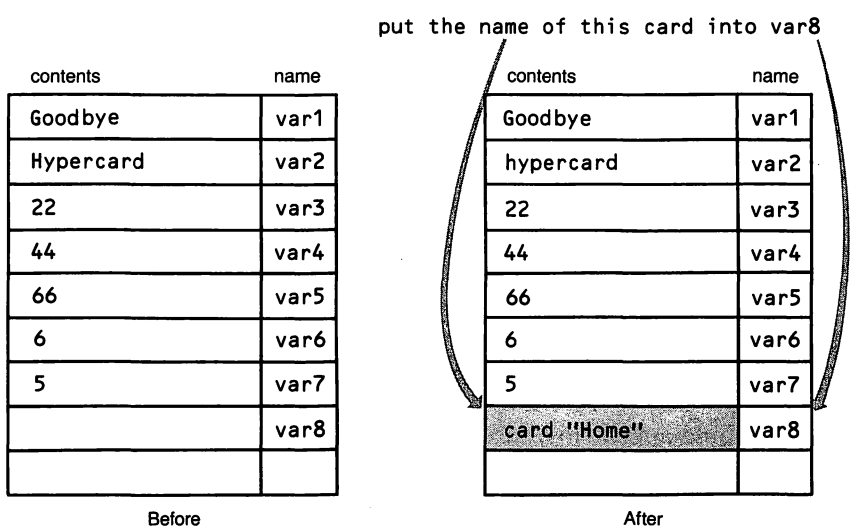


Figure 4-7. Assigning a property value to a variable.

Figure 4-7 shows how a `put` command can create a new variable and put a value derived from a property or value into the new variable. Property values are no different than container values, so you can put the value of any object property into a container. Functions like `the number of cards`, `the rectangle of button 2`, etc., also return values that can be stored in containers. In fact, you will frequently use properties and functions as sources of value for containers in your scripts. Just how frequently will become clear as we introduce more and more of HyperTalk's properties and functions in later chapters.

Containers

The final source of values in HyperTalk are the containers. You used containers in Example 4, where two variables were added together and the resulting new value was put into another variable.

As we said before, a HyperTalk container is simply a place to store a value. The complete list of containers is fields, variables, the Message box, and `the selection`. Once a container is assigned a value, that value can be referred to with the container's name. The container name stands for the value in much the same way a desktop icon stands for the document or application it represents. Which container you choose for a particular task depends on the value you need to store, how (and how fast) you need to retrieve it, and how long you need to keep it. When moving values between containers, you often call the container holding the value at the beginning *the source* and the container the value is moved into *the destination.* These terms are not HyperTalk reserved words (so you won't find them in a script), but they are often used in documentation.

Object Containers: Fields

In chapter 1, we looked at *fields* as screen objects that contain editable text. From the point of view of the HyperCard author, fields are passive things and the only advantage to putting text into a field, instead of typing it as Paint Text, is that the field can be reopened and edited. The author also knows that words in a field automatically readjust if the shape of the field changes or if the font or size of the text is changed. The scriptor, on the other hand, knows about the field's secret container identity and can use the field in a much more active way. HyperTalk can manipulate text in a field in dozens of ways, as you will see in chapter 11, "Strings."

Figure 4-8 shows three aspects of a field: the field name (or designator) as it would appear in a script, the way the field looks on the Macintosh screen, and the actual content of the field at a given time.

Fields are long-term storage containers for values up to 30,000 characters. Values are assigned to a field with a `put` command or by typing into the field. The following example demonstrates how values can be manipulated with reference to their containers.

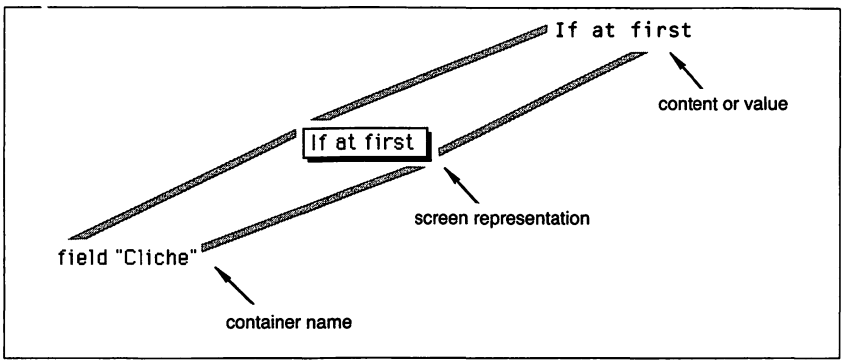


Figure 4-8. Field containers.

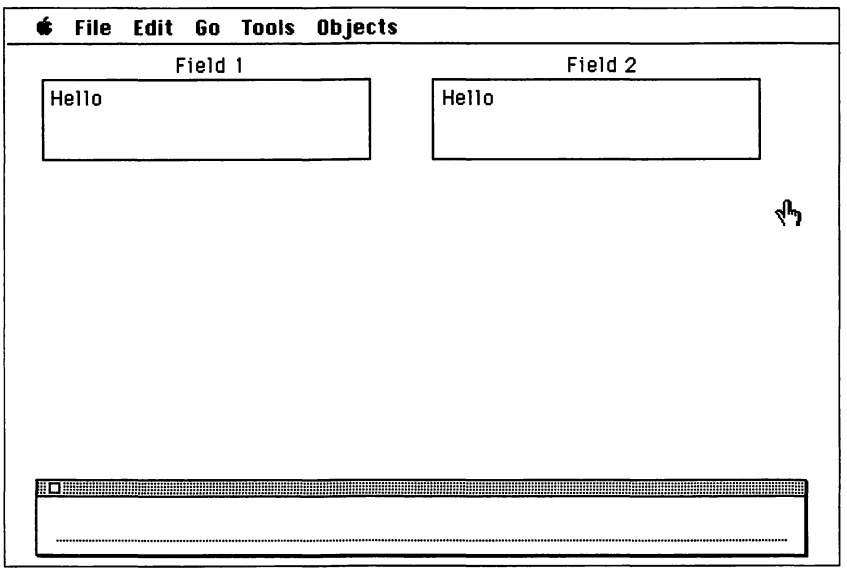


Figure 4-9. Field layout.

Example 8:

1. Go to the Audio-Visual card of your HyperTalk Bible stack
2. By selecting New Background from the Objects menu, add a new card with a new background.
3. Name the new card `Values`.
4. Add two new background fields to the Values card.
5. Place the first field on the left and the second field on the right (see Figure 4-9). Set the style of both fields to rectangle.
6. Enter the following statement in the Message box and press the Return key:

```
put "Hello" into field 1
```

Result:

`Hello` appears in the left-hand field.

7. Enter the following statement in the Message box and press the Return key:

```
put field 1 into field 2
```

Result:

`Hello` appears in the right-hand field.

Strings can be put into any container, and then that container can be put into other containers. Because the maximum size of a field is 30,000 characters, there is plenty of room in a container for even the longest string. Of course, only the value contained in the field—not the field itself—is put into the other field. Note that putting the value contained in one container into another container doesn't remove the value from the first container; it is more like copying than moving. This is a nice example of having your cake and eating it, too.

8. Enter the following statement in the Message box and press the Return key:

```
put "1234" into field 1
```

Result:

`1234` appears in the left-hand field.

9. Enter the following statement in the Message box and press the Return key:

```
put 5 + field 1 into field 2
```

Result:

`1239` appears in the right-hand field.

This shows that HyperTalk really can interpret as the number *1234* the string "1234" stored in a field. It is obviously absurd to add the number 5 to the object field 1, but HyperTalk interprets card field 1 to mean the value inside that container. Because an arithmetic operator (+) is used, the value ("1234") is interpreted as a number *1234*.

Referring to field objects that are not on the current card but are in the current stack requires that you specify exactly where the field is but doesn't require that you go to the card.

Example 9:

1. Open your HyperTalk Bible stack. If you have altered the fields on your Home card, you should make a fresh copy from the original disk for this example and restart with the "mint condition" Home.
2. Add a new card field to the first card and name it `Distant`.
3. Type the following into the Distant field.

```
"Watson, come here. I want you."
```

4. Go to the Values card, enter the following command in the Message box, and press the Return key.

```
put card field "Distant" of card one into field 2
```

Result:

The words, `Watson, come here. I want you.` appear in the right-hand field.

5. Add a new card button to the Values card and name it `Fetch Home`.
6. Enter the following handler in the Fetch Home button's script:

```
on mouseUp -- Fetch Home button
  push card
  go home
  get card field 1
  pop card
  put it into field 2
end mouseUp
```

7. Click the Fetch Home button.

Result:

The Home card is briefly displayed before the screen returns to the Values card. On return, the time (which should indicate when you last were at the Home card) in the format 6:21 PM appears in the right-hand field.

Don't Use Numbers for Names

When naming cards, buttons, or fields, don't use all numeric characters for the name. If you tell HyperTalk to locate button 2001, for example, it looks for button *number* 2001, which probably won't exist in your stack. If you must use many numbers, add some text to them, such as year2001.

In this example, the script navigated to another stack (Home), plucked a value from a field in that stack with the `get` command, which placed the value into the variable `it` (we explain `it` in a moment), and then returned to the starting card. The script then copied the value into the variable `it` in the right-hand field.

The worst part of using field containers is that you not only have to name the field you want, but you also have to name the card that the field is on if the field is not on the current card. If the field is not on the current stack, you also have to navigate to another stack. The best part of using fields is that values stored in fields are safe on the disk and are still there the next time you open HyperCard. Variables are just the opposite; you don't have to navigate to them, but they can't hold a value between HyperCard sessions. You can demonstrate this by quitting and then restarting HyperCard. When you reopen your HyperTalk Bible stack, you find `Hello` and `6:21 PM` (your time will vary) in the fields as before, but if you type `var1`, `var2`, etc., into the Message box, the values placed in those variables are gone.

Remember that fields can be hidden with the `hide` command, so values can be stored in fields without either the value or the container being visible to the user. This means that you can put values you want to preserve between HyperCard sessions into fields without affecting the appearance of the stack. In fact, you might want to use hidden fields extensively when you have limited RAM.

Variables as Containers

The classic computer language container is the *variable*, and HyperTalk boasts two versions of these valuable entities. Local and global variables have in common their

capability of containing a value and their invisibility. You never see a variable, only its name as it appears in a script. A variable's value is a character string, which can be of a length limited only by RAM.

Another nice thing about variables is that there is, for all intents and purposes, no limit to the number of variables you can use in a script. Of course, if you start dumping long text passages into variables, you run out of RAM (how soon depends on how much RAM you have), but if you want to hold a number here and a sentence there, you have no problem. Compare Figure 4-10 with Figure 4-8 to see how variables differ from fields.

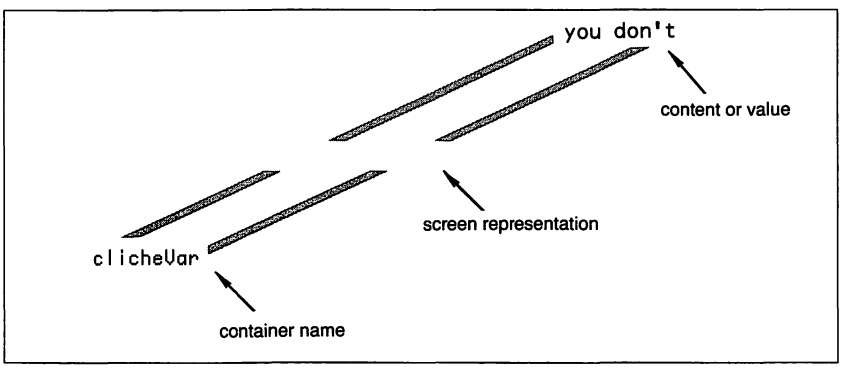


Figure 4-10. The variable container.

A variable name must start with an alphabetical character, can be of any length, and can include any of the alphanumeric characters (letters and numbers, no punctuation) as well as the underscore (_) character. One convention for variable naming uses all lowercase letters except for the first letter of words that follow the first word. Some examples are `myVariable`, `yourVariable`, and `VariableContainerBob`. Another convention separates words with the underscore character: `net_income`, `adj_gross_income`. The following are also valid variable names: `it`, `this`, `var4`, `newvalue`, `LASTNAME`. The name `var 4` is not valid because of the space, and `34Var` is not valid because it starts with a numeric character. Table 4-1 contains a summary of these conventions.

Table 4-1. Variable names.

Good Names	Bad Names	Problem Description
startingTime	start.Time	Can't use period
macVar3	3rdMacVar	Can't start with number
sanFranTime	San FranTime	Can't use space
v	8	Starts with number
retail_cost	factory-cost	Hyphen rather than underscore

It is also wise to avoid HyperTalk reserved words (HyperTalk's own vocabulary), like `button`, `mouseUp`, `beep`, and `HyperCard`, because they can cause a good deal of confusion to HyperCard (and even more confusion to you and anyone else trying to read your scripts). You can use reserved words if you combine them with other characters or words (`myStack`, `button4`, `teamHyperCard`).

A variable is assigned a value with the `put` command. Before a variable can be read (used in a math operation, for instance), it must contain a value, even if only the value `empty`.

Declaring Variables

In case you think we forgot something, we state explicitly that HyperTalk variables do not have to be declared prior to use. The only exceptions are the global variables covered in the next section, but even global variables can accept any type of value, so you can put a number into a variable one time and a string into it the next without creating any problems. As far as HyperTalk is concerned, all values are created equal, and no "advance reservations" are required.

Global Variables

So far, our variables have all been assigned values in the Message box, which has made them global variables. There are two flavors of variables in HyperTalk: global and local. *Global variables* hold their values after the handler that assigned them ends. With *local variables,* on the other hand, as soon as a handler ends, the local variable loses its value. In all other ways, all variables are alike.

As we've mentioned before, global variables actually hold their values until HyperCard is shut down or suspended. This relative immortality makes global variables ideal candidates for holding information that you need quick access to in several different handlers, or even in one handler at different times. You can think of a global variable as being shared by several different handlers.

What makes a global variable global is the `global` command, the syntax of which is shown in Figure 4-11.

```
global variableName [, variableName,variableName]
```

name of variable

list of variable names separated by commas

Figure 4-11. The `global` command syntax.

If you include the statement `global thisVariable` in a handler, the variable `thisVariable` is forever after (at least for the rest of this HyperCard session) global and can be accessed in any other handler that also contains a `global thisVariable`

statement. As you have seen, all variables assigned a value in the Message box are also global.

Example 10:

1. Open the Values card of your HyperTalk Bible stack.
2. Add a new button to the Values card and name it `Global`.
3. Enter the following handlers in the Global button's script:

```
on mouseUp -- Global button
  global Darkmoon
  put Darkmoon into the message box
end mouseUp

on mouseDown -- Global button
  global Darkmoon
  put "Sir Garth" into Darkmoon
end mouseDown
```

4. Click the Global button.

Result:

The words `Sir Garth` appear in the Message box.

In this example, the mouseDown handler runs when the mouse button is pressed, which causes the quoted string `"Sir Garth"` to be assigned to the global variable `Darkmoon`. When the mouse button is released, the mouseUp handler runs, the variable is declared global, and the value of the variable is passed to the Message box.

Following is another example that accesses a value in one handler that was assigned previously.

Example 11:

1. Open the Values card of your HyperTalk Bible stack.
2. Add a new button to the Values card and name it `Global Beep`.
3. Enter the following handler in the Global Beep button's script:

```
on mouseUp -- Global Beep button
  global var7
  beep var7
end mouseUp
```

4. The numeric value 5 (or equivalent reflecting the number of cards in your HyperTalk Bible stack) should still be in `var7` from Example 6; if it isn't, restore it with the Message box command `put the number of cards into var7`.
5. Click the Global Beep button.

Result:

Five fast beeps.

A global variable (any variable, actually) is not limited to 30,000 characters, as fields are, so it can contain even larger values than a field can, depending on your RAM. Because variables are held in RAM, they can be accessed even faster than a

disk-stored field can. Variables also are accessed with a single-word descriptor rather than a complex field descriptor, which may include a card descriptor as well. There are times when you will have values in global variables that you will want to use again the next time you use HyperCard. Because the values in variables are lost when HyperCard quits or is suspended, you need to move these values into a field or into several fields where they can be saved to disk. HyperTalk provides three messages that tell you it may be time to save your global variables to disk: `closeStack` when the current stack closes, `suspend` when HyperCard launches another application, and `quit` when HyperCard quits. In chapter 6, "Messages," we show you how to place a handler for these messages so that you can save your data before it disappears into RAM limbo.

Before we move to the next type of variable, here's a final example that permits you to play with the values of two global variables you've used before.

Example 12:

1. Open the Values card of your HyperTalk Bible stack.
2. Add two new background fields to the Values card. These will be fields 3 and 4. Position and label these fields as shown in Figure 4-12.

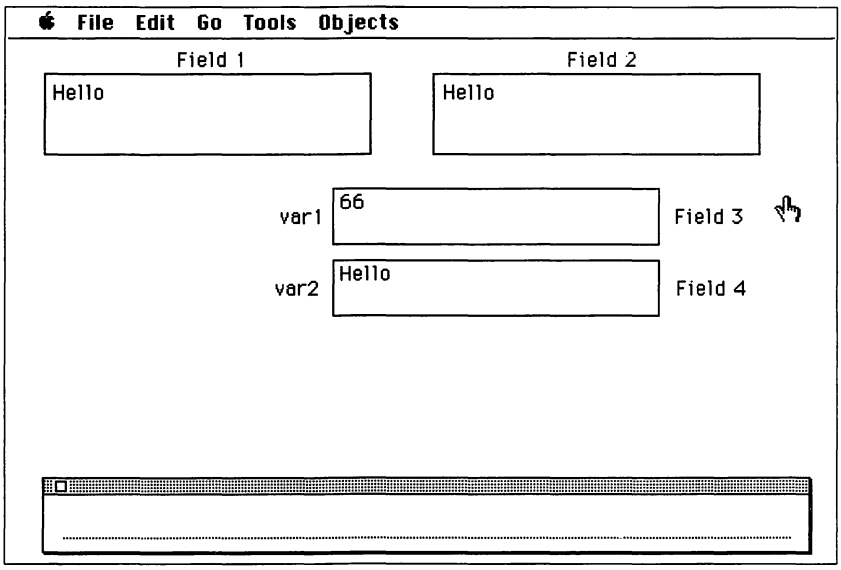


Figure 4-12. Field layout for variables.

3. Put the following handler into the card script of the Values card:

```
card script:
on idle -- card handler for fields below
  global var1,var2
  put var1 into field 3
```

continued

```
  put var2 into field 4
end idle
```

Result:

As soon as the Browse mode resumes, `Goodbye` appears in field 3, and `hypercard` appears in field 4.

4. Enter the following command in the Message box and press the Return key:

```
put 66 into var1
```

5. Enter the following command in the Message box and press the Return key:

```
put "Hello" into var2
```

Result:

66 appears in field 3 and `Hello` appears in field 4.

Globals Are Not Limited to 30K

Although fields can contain a maximum of 30,000 characters, global variables are limited by the amount of RAM on your computer. Note that Mac Plus with 1 megabyte of RAM and HyperCard running in 750 kilobytes leaves precious little RAM for the Finder, let alone HyperTalk globals. On a 2 megabyte Mac SE, global variables begin to offer more storage capacity than fields.

HyperCard sends `idle` messages whenever it isn't doing anything else. On a Mac Plus, an `idle` comes about every five ticks. The idle handler in this example constantly checks the value of `var1` and `var2` and puts their values into the two new fields. Now you can see at all times what the current value of these two variables are. Try putting new values into the variables and watch the results.

Local Variables

Unlike all the variables you've seen so far, local variables are transitory things that only "live" as long as the handler they exist in is executing. As soon as the handler ends, the contents of the variable are lost.

Example 13:

1. Open the Values card of your HyperTalk Bible stack.
2. Add a new card button to the Values card and name it `Local`.
3. Enter the following handler in the Local button's script:

```
on mouseUp -- Local button
  put "The Spoiled One is singing in the rain!" into var7
  put var7 into field 1
end mouseUp
```

4. Click the Local button.

Result:

`The Spoiled One is singing in the rain!` appears in the left field.

5. Type the following command into the Message box and press the Return key:

```
put var7 into field 2
```

Result:

5 appears in the right field.

The text value in the local variable `var7` was lost when the mouseUp handler ended, but the numeric value in the global variable `var7` remained, and it is this value that you see in the right field.

The transitory nature of local variables may seem like a serious shortcoming, but actually it is a virtue. For one thing, it allows you to use the same variable (actually several variables with the same name) in many different handlers without creating a conflict.

In the case of subroutine handlers (handlers called from within a handler that is temporarily suspended), you can use local variables with the same name at the same time in multiple handlers without a conflict.

Example 14:

1. Open the Values card of your HyperTalk Bible stack.
2. Edit the mouseUp handler in the Local button as follows:

```
on mouseUp -- Local button
  put "The Spoiled One is singing in the rain!" into var7

  -- new line below, program flow jumps to other handler
  send subroutineMessage -- new line
  -- program flow resumes here when other handler finishes

  put var7 into field 1
end mouseUp
```

3. Add the following handler to the Local button script:

```
on subroutineMessage -- Local button
  put "The Spoiled One is out with a droog!" into var7
  put var7 into field 2
end subroutineMessage
```

4. Click the Local button.

Result:

`The Spoiled One is singing in the rain!` appears in the left field and `The Spoiled One is out with a droog!` appears in the right field.

In Example 14, we used the `send` command to interrupt one handler and run another one. Even in this situation, the two identically named local variables are completely separate, as demonstrated by the fact that the left field ends up with the value assigned to `var7` at the beginning of the mouseUp handler. The value assigned to `var7` in the subroutineMessage handler does not carry over to the other handler. The `send` command is formally introduced in chapter 6, "Messages," but as you can

see from Example 14, it is used to send a message from a script that can then be handled like any other message.

The ability to use local variables in a handler without having to think about whether or not the variable name is used elsewhere saves you innumerable headaches. HyperCard's combination of local and global variables gives you the ability to build discrete routines using local variables where you want to avoid conflicts and global variables where you want to share values between handlers.

BASIC versus HyperTalk Variables

If you are familiar with Microsoft BASIC's subprograms, you can appreciate the usefulness of having variables that are local to a handler. The primary advantage of BASIC's subprograms is that their variables are local. HyperTalk's global variables largely replace the awkward parameters you have to pass to a subprogram in BASIC (though you can pass parameters to a HyperTalk handler, too). Another HyperTalk advantage is that you can call other handlers from a subroutine handler, which you can't do with a BASIC subprogram.

Just because the variable `Darkmoon` is global in one handler doesn't mean that you can't have another local `Darkmoon` variable in a second handler, provided that the second handler doesn't contain a global `Darkmoon` statement.

Example 15:

1. Open the Values card of your HyperTalk Bible stack.
2. Add a new card button to the Values card and name it `Local Dark Moon`.
3. Enter the following handlers in the Local/Global button's script:

```
on mouseDown -- Local/Global button
  put "Nowebo" into Darkmoon
  put Darkmoon into field 1
end mouseDown

on mouseUp -- Local/Global button
  global Darkmoon
  put Darkmoon into field 2
end mouseUp
```

4. Click the Local/Global button.

Result:

`Nowebo` appears in the left field and `Sir Garth` appears in the right field.

The value `Sir Garth` was assigned to the global variable `Darkmoon` in Example 10. In the mouseDown handler, you assign a new value to the new local variable `Darkmoon`. You then use a `global Darkmoon` command in the mouseUp handler to connect with the global value (sort of like getting in touch with the Power). Creating a local variable with the same name as a global variable in a handler like mouseDown without a `global` command has no effect on the global variable value.

The Local Variable it

There is a special variable called `it` that HyperTalk sometimes uses for its own purposes. You saw in the last chapter, for example, how the `answer` and `ask` commands return the user response in the `it` variable. The `get` command also places the gotten value into `it` as do the `read` and `convert` commands covered in future chapters. Because the `it` variable is so volatile and to prevent an important value from being replaced by some other value before you have a chance to use it, we recommend you immediately put the `it` value into a container that can't be accidentally changed. Any other variable or field will do.

The get *Command and* it

We've already used the `get` command once in this chapter, but we waited until now to give it a full introduction because it is so closely related to the variable `it`. The syntax for `get` is shown in Figure 4-13.

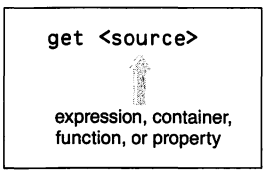


Figure 4-13. The `get` command syntax.

Quite simply, `get` is used to load values into `it`. The `ask`, `answer`, and `read` commands also use `it`, but `it` is just a means to an end for those commands. In the case of `get`, however, `it` is everything. The next example uses `get` to place the value of an object property into `it`.

Example 16:

1. Open the Values card of your HyperTalk Bible stack.
2. Type the following statements into the Message box and press the Return key for each:

```
get the name of this card
put it into field 1
```

Result:

`card "Values"` appears in the left field.

it *in Action*

Example 17 uses variables to replace and expand one of the exercises from the last chapter.

Example 17:

1. Open the Values card of your HyperTalk Bible stack.
2. Add a new card button to the Values card and name it `Music`.
3. Enter the following handler in the Music button's script:

```
on mouseUp -- Music button
  answer "Play which voice?" with "boing" or"harpsichord"
  put it into voiceVar -- line 1
  ask "What tempo?" with "150" -- line 2
  put it into speedVar -- line 3
  ask "What notes? (a-g or 20-110)" with "40 50 70" -- 4
  put it into notesVar -- line 5
  play voiceVar tempo speedVar notesVar -- line 6
end mouseUp
```

4. Click the Music button.

Result:

A series of dialog boxes is displayed, and then the notes you specify are played at the speed and with the voice you choose.

Let's look at this handler more carefully. In the first statement, the string value in the button you select is placed into the local variable `it`. In the second line, that value is assigned to the local variable `voiceVar` for safekeeping. In the third line, the value in `it` is replaced by whatever number you typed into the dialog box field. That value is transferred to the variable `speedVar` in the fourth line. This procedure is repeated again in lines five and six with whatever letters or numbers you typed into the dialog box field ending up in the local variable `notesVar`. In the last line, the three variables `voiceVar`, `speedVar`, and `notesVar` are used with the `play` command.

Note that if the voice value hadn't been saved in line two, it would have been lost when line three put a new value into it. You could actually use the variable `it` rather than `notesVar` in the last line and skip line six, but this is a bad habit to get into. It doesn't make the handler any easier to read, either.

This one handler lets you play any notes at any speed with any voice. In this case, there is no reason to retain the values contained in these variables, so we have left them local, but there may be occasions when you want to preserve those values in global variables for use at another time or in another handler.

In the previous section, we said that the special variable `it` is a local variable. Well, its default character is local, but if you give `it` a global declaration, `it` can become a global variable.

Example 18:

1. Open the Values card of your HyperTalk Bible stack.
2. Add a new card button to the Values card and name it `Global it`.
3. Enter the following handlers in the Global it button's script:

```
on mouseDown -- Global it button
  global it
  get the short name of me
end mouseDown
```

continued

```
on mouseUp -- Global it button
  global it
  put it
end mouseUp
```

4. Click the Global it button.

Result:

The words `Global it` appear in the Message box.

We included this example to show you that the `it` variable is really no different than any other variable, except that `it` is used automatically with certain commands. Although you can convert `it` to a global variable and still use `it` locally (in handlers without a global declaration) without affecting the global `it` value, we still recommend that you use `it` with caution both in its local and global forms.

To make it easier to understand what is in the `it` variable, let's add another field to the Values card to display the `it` value.

Example 19:

1. Open the Values card of your HyperTalk Bible stack.
2. Add another background field to the Values card. This becomes field 5. Position and label this field as in Figure 4-14.

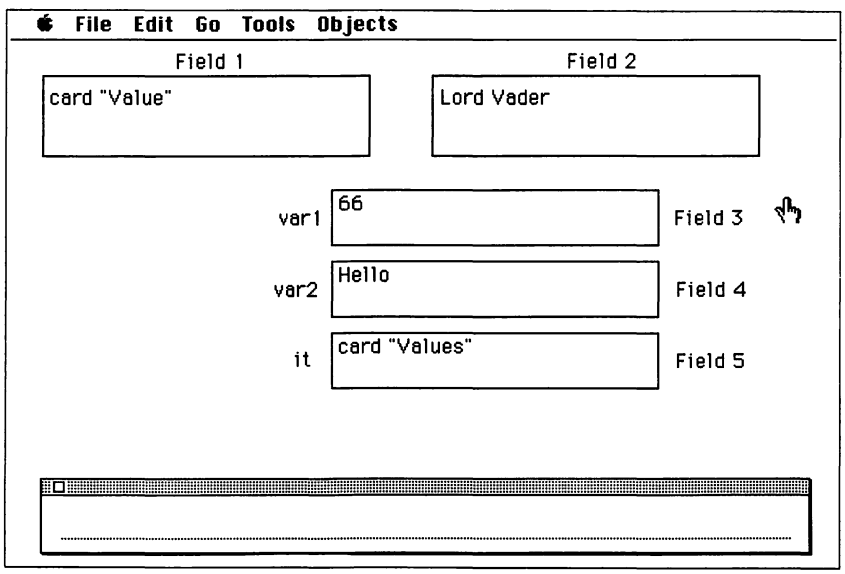


Figure 4-14. `it` demonstration layout.

3. Edit the idle handler in the card script as follows:

```
card script:
on idle -- card handler for fields below
```

continued

```
    global var1,var2,it -- changed line
    put var1 into field 3
    put var2 into field 4
    put it into field 5 -- new line
end idle
```

Result:

As soon as the Browse mode resumes, `Global it` appears in field 5.

4. Type the following command into the Message box and press the Return key:

```
get the name of this card
```

Result:

`card "Value"` appears in field 5.

Now you always know what value is in `it`. Try changing the `it` value with other `get` or `ask` commands and watch the results.

***it* Is a Local System, Not Global, Variable**

The variable `it` is a local system variable, which explains why, when you run a handler using `it`, and then type `it` into the Message box, you get an error rather than the contents of `it`. You can, however, force `it` to be a global variable by placing `it` after a global declaration in the handler you are using.

the selection as a Container

Another container, `the selection`, is a way of referring to any text selected by dragging or double-clicking with the mouse. The selection is any highlighted characters in a text field or in the Message box. This means that you can select some text and then manipulate that unit either by putting its value into another container or by putting some new value into it. See Figure 4-15 for an illustration of `the selection`. The `the` in `the selection` is optional.

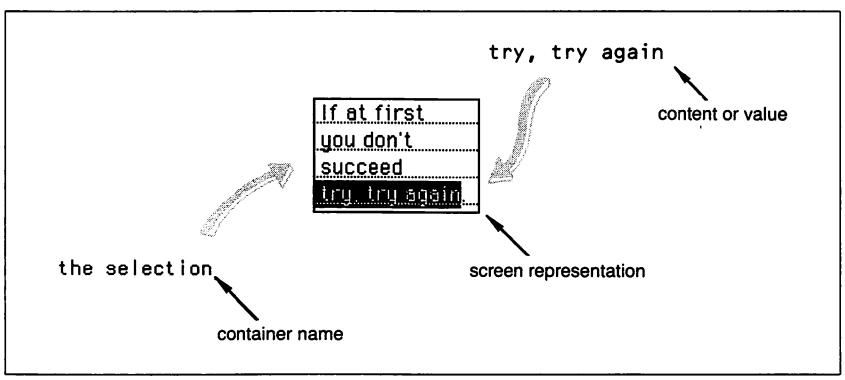


Figure 4-15. `the selection` container.

Example 20:

1. Open the Values card of your HyperTalk Bible stack.
2. Type the soul-stirring phrase, `"Far away in a long ago galaxy..."` into the left-hand field.
3. Type `"in an alternative universe"` into the right-hand field.
4. Add a new card button to the Values card and name it `Selection`.
5. Enter the following handler in the Selection button's script:

```
on mouseUp -- Selection button
  put the selection into the message box
  put field 2 into the selection
end mouseUp
```

6. Select `in a long ago galaxy` by dragging the mouse through it.
7. Click the Selection button.

Result:

The words `in a long ago galaxy` appear in the Message box, and then the words `in a long ago galaxy` in the left-hand field are replaced by `in an alternative universe`.

With `the selection`, you can mix information in a field, select just the portion you want, and use `the selection` to manipulate just that part. For instance, you could select a phone number, and then click a button that would dial the number.

Let's add a final field to the Values card to help keep track of the selection.

Example 21:

1. Open the Values card of your HyperTalk Bible stack.
2. Add another background field to the Values card and name it `6`. Position and label this field as in Figure 4-16.

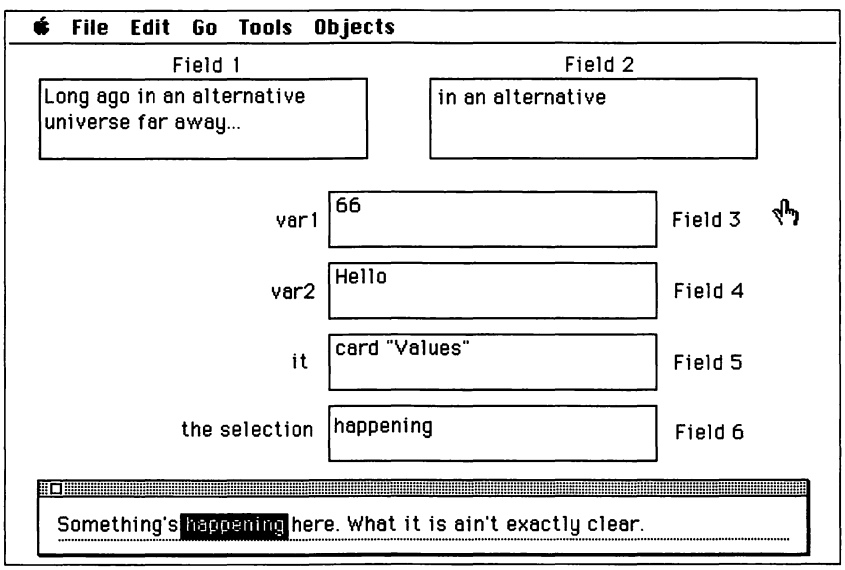


Figure 4-16. `the selection` demonstration layout.

3. Edit the idle handler in the card script as follows:

```
card script:
on idle -- card handler for fields below
  global var1,var2,it
  put var1 into field 3
  put var2 into field 4
  put it into field 5
  put the selection into field 6 -- new line
end idle
```

4. Type the following line into the Message box and press the Return key:

```
"Something's happening here. What it is ain't exactly clear."
```

5. Double click the word `happening`.

Result:

`happening` is selected in the Message box and appears in field 6.

The selection field works best when `the selection` is in the Message box, but it works after a fashion in the other fields as well. Try it and see.

Selection Functions

Four new functions that return information about `the selection` were added to HyperTalk with Version 1.2. These functions are `the selectedField`, `the selectedLine`, `the selectedChunk`, and `the selectedText`. The result of each of these functions is shown in the following list:

Related Functions:

```
the selectedFields
   Result:      bkgnd field 1
the selectedLine
   Result:      line 1 of bkgnd field 1
the selectedChunk
   Result:      chars 10 to 21 of bkgnd field 1
the selectedText
   Result:      in a galaxy
```

Previously, you couldn't tell which field `the selection` was in, much less what chunk of the field it was. All that is changed in Version 1.2 and later with the introduction of the new functions.

`the selectedField` returns the name of the field that contains `the selectedText` in the format `bkgnd field 1`. `the selectedLine` returns both the line number and the field identification in the format `line 1 of bkgnd field 1`. (If a long line is put into a narrow field or a field is made smaller, the number of lines ending in carriage returns may differ from the number of lines visible on the screen.) `the selectedChunk` returns the characters that are boxed and the field identification in the format `chars 10 to 21 of bkgnd field 1`. `the selectedText` is redundant; it returns the same value as `the selection`.

the selectedLine and the selectedChunk functions will come in handy after you pick up some knowledge of string manipulation in chapter 11, "Strings", but you can use the selectedField function now to get the context of the selection. Example 22 puts the entire string, of which a selection is a part, into the Message box.

Example 22:

1. Go to the Values card of your HyperTalk Bible stack.
2. Edit the handler in the Selection button's script as follows:

```
on mouseUp -- Selection button
  put the selectedField into tempVar
  put the value of tempVar into the message box
end mouseUp
```

3. Select in an alternative universe by dragging the mouse through it.
4. Click the Selection button.

Result:

The words Long ago in an alternative universe far away... appear in the Message box.

We've used another new function in this last example called the value. When a container holds the name of an object, as tempVar does in this case, putting the container would move the name of the object rather than the contents of the named object. Because we want to display the contents of the field, we specify the value of tempVar, which actually gives the value contained in the object whose name is the value in tempVar. Run Example 22 again with the value of deleted, and you see the field name rather than its content displayed in the Message box.

These new functions give you improved control over what you know about the selection, which translates into greater flexibility and power.

The Message Box as a Container

Our last container is the Message box. Its value can be manipulated the same as any other container. The Message box is an unusual container that can hold a value of any length as long as it is all on one line. The problem here is that the Message box operates by looking for return characters and executes whenever it sees one. Because return characters separate lines, you can't have more than one line without also having a return character. Unfortunately, the Message box is not scrollable, so if you put a long line into it, you may not be able to see it all. Nevertheless, if the line is a valid statement, it runs as expected. Figure 4-17 illustrates how the Message box looks on the screen and in its script.

Values are assigned to the Message box from a script with the put command. You can also have the user type directly into the box prior to running a script, or you can type functions, properties, and commands into the Message box as we have been doing in many of the examples in this book. There are a number of ways to refer to the Message box in a script: the message box, message box, the message window, mes-

sage window, the msg, and msg are all valid designators. Indeed, the Message box is the default destination for the put command when no destination is specified, so the statement put "Hello" displays the string Hello in the Message box. Although this is handy at times, keep in mind that shortcuts like this make your scripts very hard to read. In this book, we always try to use at least msg when we are referring to the Message box.

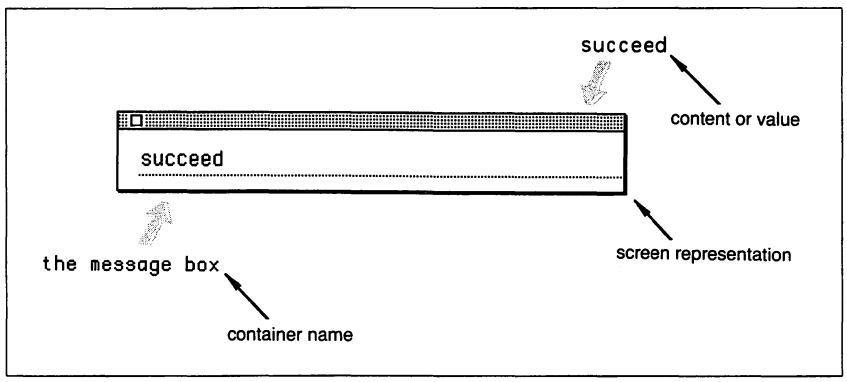


Figure 4-17. The Message box container.

We have already used the Message box as a container for various values. As you recall, the process goes something like this.

Example 23:

1. Open the Values card of your HyperTalk Bible stack.
2. Enter the following put command in the Message box and press the Return key:

```
put the id of this card
```

Result:

card id 2312 (you will get a different number) is displayed in the Message box.

3. Type "Let me take you down 'cause I'm going to Strawberry fields. Nothing is real" into the left field (field 1). Be sure to let the text wrap naturally at the end of the line; don't type a Return character. (To get rid of the previous text in field 1, put empty into the field or backspace/delete through it.)
4. Enter the following in the Message box and press the Return key:

```
put field 1 into message window
```

Result:

Let me take you down 'cause I'm going to Strawberry fields. Nothing is is displayed in the Message Box.

However, we have yet to reach out and grab the contents of the Message box. Let's do that now.

Example 24:

1. Open the Values card of your HyperTalk Bible stack.
2. Add a new button to the Values card and name it `Get a Message`.
3. Enter the following handler in the Get a Message button's script:

```
on mouseUp -- Get a Message button
  ask "Type a message..." with "here."
  put it into message box
  answer message box
  put msg into field 1
end mouseUp
```

4. Click the Get a Message button.

Result:

A dialog box prompts you to type a message. When the dialog box closes, the message is displayed in the Message box, which is used to provide the prompt string for an answer box. For good measure, the Message box value is then put into a field.

As we said before, all variables used (assigned a value) in the Message box are global, so you don't have to use the `global` command to read their values or assign them new values in the Message box. This automatic global feature can save you some time when you are debugging a handler and want to set the values of any global variables prior to the execution of the handler. Just assign the values in the Message box and execute the handler. You can also learn the current value of a global variable (`weirdVar`, for example) at any time by simply typing `put weirdVar` into the Message box. If `true` is displayed when you expected a number, you know you have a problem. You can start looking for where `true` could have sneaked in. We've used the Message box to read variable values in this chapter, and you can continue to use it as you work with HyperTalk.

Option-Tab Tears Off Tools Windoid

If you get tired of tearing the tools windoid off the menu bar, you can use option-tab, which tears it off for you. If the tools windoid has a paint tool selected, pressing Tab will tear off the patterns windoid for you.

Summary

Variables are containers represented by names that stand for the values they contain. Variables are used as temporary storage for all kinds of values: for values that determine program flow, for values that are passed as parameters with command messages, for values representing intermediate steps in calculations, for values that must be manipulated or tested before being used, and for values that must be readily available at all times. HyperTalk supports two different kinds of variables: global variables, which preserve values in memory after a handler ends so that the value

can be accessed by other handlers that contain a global declaration for the variable, and local variables that immediately forget the value as soon as the handler ends.

We covered the special variable `it` and saw how it is used with `get` and other commands to put values into a convenient work space so that we can manipulate them. In addition to variables, which are common to all programming languages, HyperTalk provides three other kinds of containers: fields, `the selection`, and the Message box. Fields are the screen objects you are familiar with from your days as a HyperCard author. `the selection` consists of any text in fields or in the Message box that has been selected. The Message box is the one-line windoid that pops up at the bottom of the screen when you do a `find`. With these four options, you can find the proper container for almost all your value storage needs.

Review Questions

1. What is the only HyperTalk container that preserves a value between HyperCard sessions or during a suspension?
2. Name a HyperTalk container that can have only one line.
3. Name a HyperTalk container that you can't see on the screen.
4. What is the remaining HyperTalk container type?
5. Which of the following variable names are valid, which are invalid, and why?

   a. `variable1`
   b. `num__var`
   c. `o1345`
   d. `2ndVar`
   e. `this.Var`

6. Is `tempVar` in the following handler a global or local variable?

```
on mouseUp
  put the name of this card into tempVar
  put tempVar into message box
end mouseUp
```

7. Connect the values types on the left below with the values on the right:

a. number	e. 8
b. constant	f. nine
c. quoted string	g. "ten"
d. unquoted string	h. eleven

8. Connect the functions on the left with the values they return on the right:

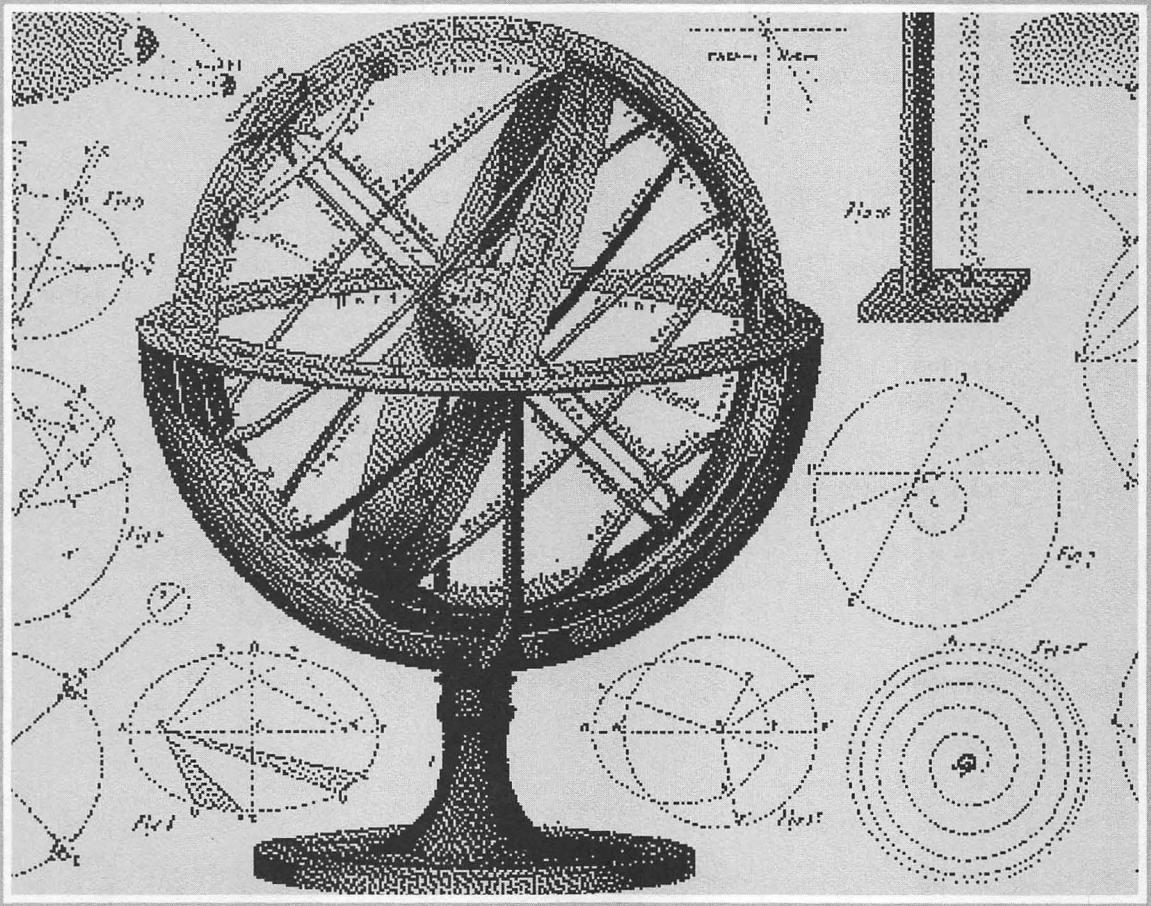
a. `the selectedField`	e. `chars 4 to 10 of bkgnd field 2`
b. `the selectedChunk`	f. `Beam me up, Scotty`
c. `the selectedLine`	g. `card field id 4`
d. `the selectedText`	h. `line 4 of bkgnd field "Tribbles"`

Exercise

Make an Add button for the Values card that uses the `add` command to put the value of `var1` plus the value of `var2` into `var2`. (The result will be displayed in Field 4.)

Chapter 5

Simple Math



Quick View

Commands
add
subtract
multiply
divide

Functions
abs
round
random
average
trunc

Operators
+, -, *, /
div
mod
^

Error Msg
NAN
INF

Property
numberformat

Concepts
Math operators
Using operators in scripts
HyperTalk constants
Shortcuts with arithmetic commands
Error messages
Advanced operators
Project: scientific calculator
Incrementing calculator
Functions as transformers
Arguments
Precedence
Size of HyperTalk numbers
Controlling number of digits
Project: chaos theory

Chapter 5

Simple Math

One day, two ancient mathematicians looked off into space, thinking about the fact that there was no way to represent zero. Suddenly, one of them jumped up, saying, "Hey, I got it, I got it." The second mathematician asked, "What, what is it?" The first mathematician gave a puzzled look, blinked, and said, "Oh, nothing."

Math in the HyperTalk language is so simple that you will wonder how you ever got along without it. What makes math so easy to understand and use is that instead of just manipulating impassive dull numbers, HyperTalk gives you the freedom to use numbers to manipulate objects, to control their properties, to place things in containers, and to control the locations and sizes of icons, fields, and graphics. With such devices available, HyperTalk lends itself to visual and audio examples of mathematical operations.

In this chapter, we explore the ingratiating numeric universe of HyperTalk, learning its operators and commands. We tell you how HyperTalk represents numbers, and we introduce four additional HyperTalk commands (`add`, `subtract`, `multiply`, and `divide`) that provide shortcuts for the scriptor. We teach you HyperTalk's 23 built-in math functions that let you do the same calculations you find on a scientific calculator, and we also show you integer division (`div`), the remainder operator (`mod`), and the exponent operator (`^`). The last three are particularly useful when controlling loops and repeats in later chapters, so make sure you understand them well. The constants `one` through `ten`, `pi`, and `true/false` are introduced, as are the five popular functions `abs`, `round`, `random`, `trunc` and `average`.

We don't put you to sleep with a dreary chapter on all the math functions—just the ones most often used. (Chapter 13, "Math Functions," makes you suffer through the more advanced functions.)

The Math Operators

On the most rudimentary (but practical) math level in HyperTalk, you enter simple math calculations directly into the Message box as you would into a calculator. Using the Message box this way, your calculations have no effect on the stack that is cur-

rently running. This gives you a practice pad, because the Message box executes any legal HyperTalk statement you type into it.

The most common use of math calculations, however, is in scripts. The results of a calculation can be stored in any container—a regular or global variable—or in a field. We will cover using calculations in scripts soon, but for now we'll use the Message box because we can instantly see results and it takes fewer keystrokes. The popular arithmetic operators (+, -, *, and /) work as expected in HyperTalk equations and scripts.

Addition in HyperTalk

Simple addition is accomplished in HyperTalk with the addition or plus operator, which is represented by +. Use it to add two simple constants and display them in the Message box, just as you would in a simple calculator.

Example 1:

1. Enter 5 + 3 in the Message box and press the Return key.

Result:

The sum 8 appears in the Message box.

If you want to do an entire new calculation in the HyperCard Message box, you must clear the display. On the Macintosh SE, and Macintosh II keyboards, there is a key called Clear that clears the previous result. The Macintosh Plus and older Macintosh keyboards lack the Clear key. HyperCard is pretty smart, however, and you don't always have to clear the Message box with the Clear key. For example, if there is a previous result in the Message box and you want to add two numbers together, simply begin typing. HyperCard recognizes that the first character you type is a number and automatically clears the previous number.

2. With 8 still in the Message box, enter + 4 in the Message box and press the Return key.

Result:

12 appears in the Message box, showing that HyperTalk can keep a running total.

HyperCard notes that the first character you type is a plus, and assumes you want to add the next number you type to the previous one. If you type another plus and another number, HyperCard continues to add the number to the previous result and displays it in the Message box. This ability to keep a running total is also true of subtraction, multiplication, and division.

You can use + to add up your grocery list. (Again, you don't have to clear the display, just start typing.)

Example 2:

Enter

```
10.12 + 4.35 + 14 + 2.00 + .10
```

in the Message box and press the Return key.

Result:

30.57 appears in the Message box.

This shows that HyperTalk handles decimal numbers as well as integers (whole numbers). Here is another example using decimal points:

Example 3:

Enter 1 + .999999999 in the Message box and press the Return key.

Result:

2 appears in the Message box.

The number 1.99999999 is displayed as 2 as if HyperTalk rounded it off. This example shows that there are some limits to how accurately HyperTalk represents decimal numbers.

Subtraction in HyperTalk

Subtraction in HyperTalk uses the minus operator - to subtract two numbers. Do subtraction in the Message box or inside a script.

Example 4:

Enter 10 - 11 in the Message box and press the Return key.

Result:

-1 appears in the Message box.

HyperTalk understands negative and positive numbers and indicates negative numbers with a minus sign in front of them. Positive numbers are assumed when there is no sign shown.

Multiplication and Division in HyperTalk

Use the asterisk * for multiplication and the slash / for division. Use the same "running calculator" feature with the multiplication and division operators as well. Suppose you want to multiply successive numbers by 2, as in 2 × 2 × 2, and so on. Type 2 * 2 into the Message box and 4 appears. Now type *2. HyperTalk automatically puts the * and the 2 in the right place (adding spaces) and completes the calculation. Striking any operator key inserts the space and the number, and performs the calculation.

Using Arithmetic Operators in Scripts

Obviously, you are learning HyperTalk not to use as a sophisticated calculator, but to help you within scripts. For example, a stack that does loan amortizations contains a script that reads various numbers entered in fields on a card, calculates the totals, and presents the results in another field. In this case, the results of intermediate calculations need to be stored. You can perform calculations on the contents of the fields in other stacks as well as in other cards. A HyperTalk calculation in a script is not much different than one in the Message box, except that you need to surround the calculations with `on` and `end` statements to trap the message you want to handle.

Use the operators in your scripts to perform various calculations on simple variables or more complex calculations involving fields and global variables. In the simple example of using just variables, a script calculates the tax on an item that the user enters. Let's say there are two variables, `amount`, which holds the amount spent, and `tax_rate` which holds the current tax value as a decimal fraction. In this case, a script contains a line something like this:

```
put amount * tax_rate into sub_total
```

Here we define `tax_rate` to be .06 and `amount` to be $100. The product of .06 and 100 is 6 and is put in the variable `sub_total`.

Operators are used with fields as well. Suppose you have a field called amount, and a simple variable called `tax_rate`. The script line

```
put card field amount * tax_rate into card field total
```

performs the calculation on the contents of the amount card field and the `tax_rate` variable and puts the result into a card field called total. Figure 5-1 shows how this works.

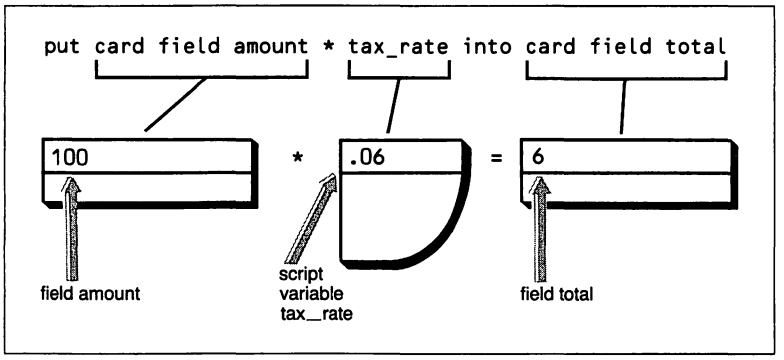


Figure 5-1. Multiplying contents of a field by a variable.

HyperTalk Constants

With HyperTalk, you may use the names of numbers ten and less as well as the numbers themselves. In other words, you may type `one + one`, or `1 + 1`. These are numeric constants. Table 5-1 lists these constants and their numeric values. There are also nonnumeric constants, such as `tab` and `up`.

Table 5-1. HyperTalk Constants

Numeric Constant	Value
zero	0
one	1
two	2
three	3
four	4
five	5
six	6
seven	7
eight	8
nine	9
ten	10
pi	3.14159265358979323846

The main use of the names is to make your scripts more *self-documenting*. This means that they are easier for someone to figure out because the script reads like an English sentence. Write

```
put feet * ten into toes
```

or

```
put eggs / six into half_dozen
```

or

```
put bytes / eight into bits
```

Unfortunately, because only the first eleven numbers are represented in HyperTalk, you can't enter the name of a number greater than ten. So, "twenty-one" or "three point one" are not acceptable. You can write clever things like

```
put two + two into you_hoo
```

or

```
put five + six into pickup_stick
```

Another useful constant in HyperTalk is `pi`. This is a convenient way to write a long and cumbersome number representing the ratio of the circumference to the diameter of a circle. Because `pi` is a long *irrational number* (it has no exact value), it can't be represented perfectly in any language. HyperTalk represents it to 21 digits:

```
3.14159265358979323846
```

You see `pi` used in script statements like this:

```
put two * pi * radius into area
```

where the formula for calculating the area of a circle uses the constants `two` and `pi`.

There are two other often-used constants: `true` and `false`. Think of them not as numbers that are used with the `+`, `-`, `*`, or `/` operators, but rather as "logical constants" that are used with the logical operators.

Arithmetic Commands: A Programmer's Shortcut

HyperTalk has a set of built-in math commands that work very much like the math operators we just learned, except for a subtle difference. These commands provide a shortcut for the regular math operators when you are doing calculations and don't care to save any intermediate results. For example, in the statement

```
put a + b into c
```

we care about all the variables, `a`, `b`, and `c`. When we add `a` and `b`, the result goes into `c`, and `a` and `b` are preserved—they still have their old values. What if you don't need to preserve the value `b`? Then, such a statement with its three variables is wasteful because it requires an unneccessary variable: `c`. Rewrite it like this:

```
put a + b into b
```

Here, we add `a` to `b`, and put the result into `b`, which essentially replaces the old `b` value with the new one. There are many applications where such a situation exists: keeping a running total of expenses, moving a ball across the screen, or increasing the value of a bar in a graph. HyperTalk provides the command `add`, which replaces the previous script with an even simpler syntax:

```
add a to b
```

Here, the word `add` means to add `a` to `b` and put the result in `b`.

HyperTalk provides a set of four commands to use when you don't care about preserving intermediate results: `add`, `subtract`, `multiply`, and `divide`. All require

you to specify a *source* and a *destination*. The source may be a container (a variable, field, Message box, selection) but the destination is always a container. The source holds the value from which the program gets the information to perform the operation, and the destination is the container in which the program puts the result of the operation. Each of the commands uses the destination and source syntax slightly differently. Figure 5-2 shows the syntax for the `add` and `subtract` commands and Figure 5-3 shows the syntax for `multiply` and `divide`. Note that the order of the source and destination are swapped for the two sets of commands.

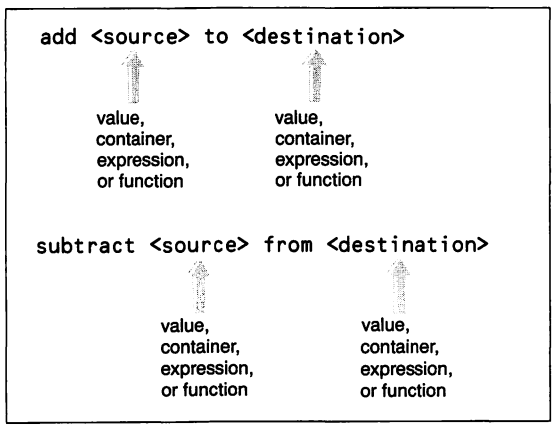


Figure 5-2. The `add` and `subtract` command syntaxes.

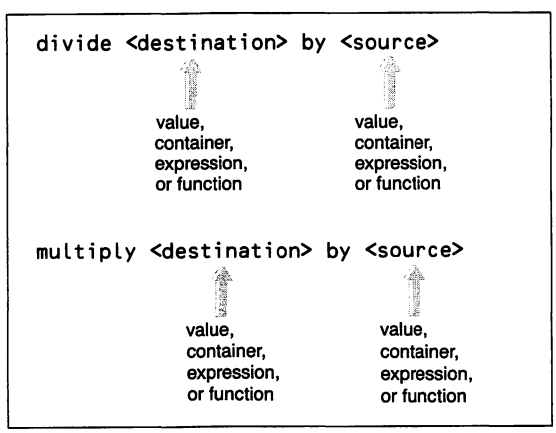


Figure 5-3. The `multiply` and `divide` command syntaxes.

The source and destination parameters may also be *expressions*. Expressions are simply a collection of HyperTalk commands and operators that can be evaluated correctly. For example, `4 / 5 * 2` is an expression. If you type the expression in the

Message box and it gives back a result, the expression will work in commands that show it as a parameter.

The *add* Command

Let's do a few examples with the `add` command.

Example 5:

1. Enter `put zero into rock` in the Message box and press the Return key.

Result:

Although nothing happens, you have just created a variable called `rock` and put the value `zero` (0) into it.
One of HyperTalk's best features is that it creates variables when you first use them without requiring the variable-type definition that other languages, such as C and Pascal, require.

2. Enter `add 10 to rock` in the Message box, press the Return key, enter `rock`, and press the Return key.

Result:

The value 10 appears in the Message box.
The `add` statement adds the value 10 to the variable `rock` and, when we type `rock` in the Message box, HyperCard reports that it is equal to 10.

3. Enter `add 10 to rock`, press the Return key, enter `rock`, and press the Return key.

Result:

The value 20 appears in the Message box.
The `add` command simply adds the value in the source, the constant 10, to the current value of destination (named `rock` and equal to 10), and puts the result back in `rock`, making `rock` equal to 20.

4. Enter `add 10 to roll`, and press the Return key.

Result:

HyperCard gives you the error message `can't understand arguments to command add`.

The result of this last example shows that the `add` operator cannot create a variable, as `put` can. Before any of the arithmetic commands can be used, the variables and containers must be created with `put`. That is why our first statement put the value `zero` into `rock`.

A typical example using the math commands supposes you are incrementing the position of a button moving horizontally across the screen. This requires repeatedly adding a set number to the horizontal value. With normal math operators, assuming the button's horizontal x location is stored in the variable called `x`, write

```
put 10 + x into x
```

With the `add` command this is simplified to

```
add 10 to x
```

The *subtract* Command

The `subtract` command works the same way the `add` command does, but with the word `from` rather than `to`. An example is

```
subtract deduction from total
```

which takes the value of `deduction`, subtracts it from `total`, and puts the result in `total`.

The *multiply* Command

The source and destination arguments for the `multiply` command syntax are reversed, and even though the destination precedes the source, the result ends up in the destination. Again, note that the destination must be a container.

Example 6:

1. Enter `put pi into chaos` in the Message box and press the Return key.
2. Enter `multiply chaos by two` in the Message box, press the Return key, enter `chaos`, and press the Return key.

Result:

`6.283185` appears in the Message box.

This shows that `multiply` works much like `add` and `subtract`.

The *divide* Command

The source and destination arguments are also reversed in the syntax for the `divide` command and, even though the destination precedes the source, the result ends up in the destination.

Example 7:

1. Enter `put ten into decade` in the Message box and press the Return key.
2. Enter `divide decade by ten * ten` in the Message box, press the Return key, enter `decade`, and press the Return key.

Result:

`0.1` appears in the Message box.

Use expressions as well as containers with these commands.

Example 8:

Enter `divide 100 by (10 * 10)` in the Message box and press the Return key.

Result:

`1` appears in the Message box.

When Not to Use the Math Commands

When shouldn't you use the math commands? Well, they don't make much sense when you need to know the difference between two containers and you wish to keep the containers' values unaffected by the subtraction. If you have a field called vessel_a and another called vessel_b, and you want to display the contents of these two fields and the difference between them, use a command like

```
put card field vessel_a - card field vessel_b into card field diff
```

The `subtract` command does not work well here and requires several separate temporary variables. The rule, which also applies to the multiplication and division commands, is: Don't use these commands when the intermediate results of your calculations are important. The intermediate results are often useful for debugging purposes, because more information about the calculation exists at one time.

Error Messages in HyperMath Land

You can make mistakes in HyperTalk math but, unlike most programming languages, HyperTalk gives you friendly error messages that not only complain lightly when you have made a mistake but do it in a way that is useful.

*INF*inity

HyperTalk was designed to be really simple and straightforward, so the way it deals with illegal operations is interesting.

Example 9:

1. Enter `1/0` in the Message box and press the Return key.

Result:

`INF` (positive infinity) appears in the Message box.

2. Enter `-1/0` in the Message box and press the Return key.

Result:

`-INF` (negative infinity) appears in the Message box.

The result `INF` stands for INFinity and is displayed when you try to divide a finite number by an infinitely small number (zero). When you divide a negative number by `0`, you get the result `-INF`. This spiffy feature of HyperTalk works in scripts too and, when a script makes a divide-by-zero mistake, it returns `INF`.

NAN: Not a Number

Another possible error with numbers is made by trying to do something that is just plain illegal, such as dividing zero by zero. If you enter `0/0` in the Message box, you

get NAN(004) back. NAN means "not a number" and the (004) is special programmer-friendly, user-hostile information that means something if you have the *Apple Numerics Manual* (published By Addison-Wesley in 1988). If you don't, just ignore the (004). You see NAN other times if you use the square root function to take the square root of a negative number, for example.

HyperTalk's Limited Error Checking
One of the benefits of having HyperTalk represent numbers as text strings is that you can easily trap these errors. Using the `if-then` statements, HyperTalk looks for the string `INF` and performs however you wish it to when the error occurs. However, HyperTalk is seriously lacking in general error messages. Errors in scripts are fatal and cause a fixed dialog box to appear on the screen. The present version of HyperTalk does not allow you to bypass or trap this mechanism.

Advanced Operators

HyperTalk offers other operators that do more advanced operations on numbers. These include `div`, `mod`, and `^` (exponent).

Whole Number Division: *div*

The `div` operator is used to divide two numbers when you don't care about the decimal remainder.

Example 10:

1. Enter `10/4` in the Message box and press the Return key.

Result:

`2.5` appears in the Message box.

2. Enter `10 div 4` in the Message box and press the Return key.

Result:

`2` appears in the Message box.

The `div` operator divides the two values and truncates the remainder of such a division. For example, if you want to know how many times 12 goes into 143, you do not want to see an answer of 11.9166667. If you use `143 div 12` the answer is 11 (the .9166667 is truncated).

You might be tempted to use this operator when you are controlling the location of an object on the screen and the pixels are only located at whole numbers. In this case, the result of a calculation that moves the object produces a useful fractional result and you actually want the nearest whole number reported. In such a case, you use the `round` operator, which does not truncate the decimal part.

The Remainder Operator: *mod*

The mod operator is not a smart aleck wearing flowered shirts and bell-bottomed pants. There are often times when you wish to know the value left over after you divide two numbers. This is particularly useful for determining whether a number is even or odd. The `mod` function is used for this purpose. (In Latin, modulus means "a little unit of measurement" but in math, it is known as a number for converting logarithms in one system to another.) The `mod` function has various uses in programming, especially for generating a special range of numbers from an existing one (in loops). This is how it works in the Message box:

Example 11:

1. Enter `10 mod 4` in the Message box and press the Return key.

Result:

The value `2` appears in the Message box.
This shows us that 10 divided by 4 has 2 left over. Try `mod` with other numbers in the Message box.

2. Enter `7 mod 2` in the Message box and press the Return key.

Result:

The value `1` appears in the Message box.

3. Enter `6 mod 2` in the Message box.

Result:

The value `0` appears in the Message box.

Steps 2 and 3 show that for the formula `n mod 2`, the result is `1` if the number n is odd and `0` if the number n is even.

The Exponent Operator: ^

The *exponential operator* is used to raise a number to a power.

Example 12:

1. Enter `2^2` in the Message box and press the Return key.

Result:

4 appears, which shows us 2 squared is 2 × 2, or 4.

2. Enter `two^ten` in the Message box and press the Return key.

Result:

1024 appears, which is 2 to the 10th power.

You can also raise a decimal number to an exponent and use a decimal number for the exponent itself: `2.5^10.7`.

Project Script: Poorman's Pseudoscientific Calculator

One way to learn more about the operators is to create a project script that lets you use them in scripts. Refer to the card in Figure 5-4 as you build the calculator. This figure shows the final version of the calculator, with all the buttons and fields to add as you complete this chapter. All project stacks in this book follow this format. The complete card is always shown—you add the gadgets and parts to finish it.

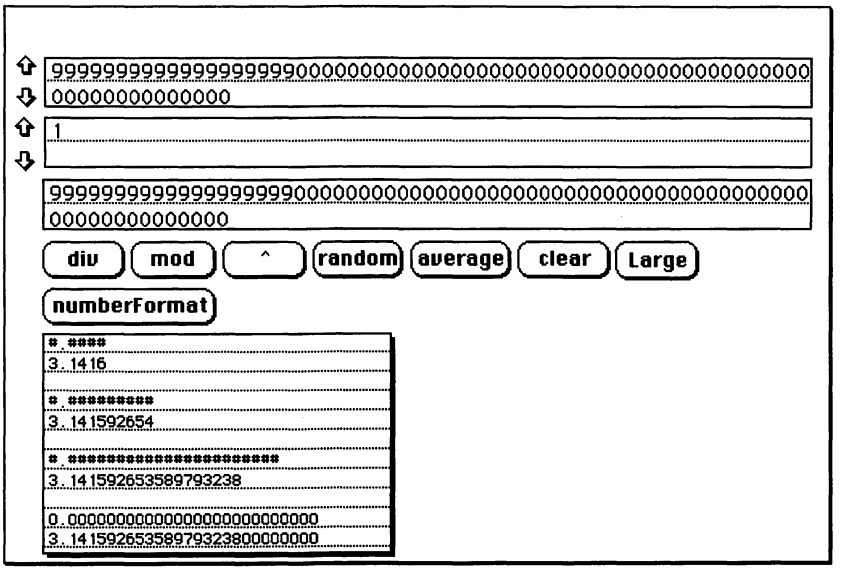


Figure 5-4. Poorman's pseudoscientific calculator.

Example 13:

1. Create a new card in your HyperTalk Bible stack, name it `Calculator`, create three new rectangular text fields, and arrange them as shown in Figure 5-4. Name the top field `x`, the field below it `y`, and the bottom field `Result`. Arrange the fields as shown in Figure 5-4, and enter small numbers (such as 5 and 2) in each of the top two fields.
2. Create three new buttons, name them `div`, `mod`, and `^` and arrange them as shown in Figure 5-4. (Ignore the other buttons in the figure for now. We will add them later when we expand the calculator.)
3. Put the following handlers into the div, mod, and ^ buttons:

```
on mouseDown -- div button
  put card field x div card field y into card field Result
end mouseDown
```

continued

```
on mouseDown -- mod button
  put card field x mod card field y into card field Result
end mouseDown

on mouseDown -- ^ button
  put card field x ^ card field y into card field Result
end mouseDown
```

4. Press the various buttons while numbers are in the top two fields, x and y.

Result:

The results of the operations appear in the Result field. Change the values in fields x and y. Try to predict the results before you click a function button.

Extra Credit: Incrementing Your Calculator

The next example shows how to get the numbers in the calculator to quickly increment or decrement, a task that takes a lot of programming code in any other language. It also shows another use of the `add` and `subtract` commands.

Example 14:

1. Add four new buttons to your card with the calculator. Make these transparent buttons, and give them the little up and down arrow icons shown in Figure 5-4 next to the top two fields. Put the following scripts into the buttons:

```
on mouseStillDown -- upper up arrow button
  add one to card field x
end mouseStillDown

on mouseStillDown -- upper down arrow button
  subtract one from card field x
end mouseStillDown

on mouseStillDown -- lower up arrow button
  add one to card field y
end mouseStillDown

on mouseStillDown -- lower down arrow button
  subtract one from card field y
end mouseStillDown
```

2. These are now increment buttons for the numbers in the fields. Click the increment and decrement buttons. Click the calculator buttons and see how high HyperTalk numbers can be and note how many digits of accuracy seem to be represented.

Result:

Watch the numbers quickly roll up and down like a fast-turning automobile odometer as you click on the increment and decrement buttons.

Functions: Think of Them as Transformers

HyperTalk has a special set of tools that are extremely useful to the scriptor. These are called *functions*.

A command, as you know by now, performs some task in HyperCard, e.g., going to another card, beeping, or selecting a menu item. Commands make something happen. Functions also perform a task but function tasks usually involve getting information from HyperCard or manipulating values in calculations. The distinctive characteristic of functions is that they return a value as a result of their execution—in effect, transform into something new. This works like the `answer` command, which returns the name of the button selected in the variable `it`.

Secret Script Debugger in Home

Not many people know it but there is a neat script debugging tool built into the Home stack of HyperCard. It's a function called `searchScript`. Use it to search for words in scripts. Give `searchScript` two parameters: a word to search for and the name of the stack to search in. Suppose you had used the variable `soundflag` in the script of hundreds of cards of the stack named Tricks. You could find every occurrence if you simply type:

```
searchscript "soundFlag", "Tricks."
```

You might think of functions as transformer toys—little robots that turn into airplanes, tanks, or racing cars. A function changes a number from one costume to another. For example, a number turns into a square root or even a letter.

Functions are not the same as commands. A function is executed the same way a command is, but a function returns a value in its place after it's executed. A function is usually fed a value, does its operation, and returns with a new value. A function is like a built-in formula that operates on the value you feed it, so, in a way, a function actually turns into the thing it is supposed to be. For example, if you type a function into the Message box and give it a value to work on, when you press the return key, the function is replaced in the Message box by its result.

A typical function is `random`. Let's see how it and the other four popular functions work.

random: The Ultimate Abstractor

HyperTalk has a lovely function that generates random numbers. Why would you want to do this? You will find that, in writing stacks, random numbers are important for controlling the random position of enemy ships in games, for simulating a statistical process in HyperCard (like random gas movements, random population swings, probability, etc.), in education and simulation, for testing purposes, for music composition, and for any process in which you want a series of unpredictable numbers. Figure 5-5 shows the syntax for the `random` function.

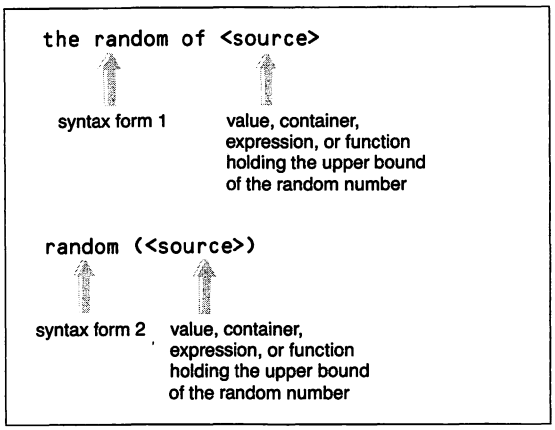


Figure 5-5. The `random` function syntax.

There are two ways to tell HyperTalk to generate random numbers. (This is true for many of the HyperTalk functions.) The first form is `the random of <source>` where `<source>` holds the upper limit of the range of random numbers that are generated. The second form is `random(<source>)`, and it requires you to surround the source of the upper limit value with parentheses. In either case, only whole numbers are produced by random. You have to play some programming games to get decimal random numbers.

Let's see how `the random` works and learn more about functions in general at the same time:

Example 15:

1. Open your HyperTalk Bible stack.
2. Enter `the random of 100` in the Message box and press Enter.

Result:

A random number between 0 and 100 appears in the Message box.

Remember, when you enter a function in the Message box and press the Return key, the function is replaced by its result. You can use the `random` function in many creative ways. Try this next.

Example 16:

Enter `play "harpsichord" 40 + the random of 40` in the Message box and press Enter.

Result:

A random note is generated. Because the entire statement stays in the Message box, you hear a different note each time you press the Return key.

This time, the Message box's contents are not replaced by the executing function when you press the Return key because HyperTalk recognizes that `play` is the chief command on the line, and knows that the `play` keyword is not a function but a

command with an expression in it. HyperTalk first evaluates the value of `the random of 40`. That creates a random number between zero and 40 that is added to the value 40 to produce a random number between 40 and 80. That value, in turn, becomes the actual note that the `play` command uses to play the harpsichord voice. By adding the constant to the note, we boost it into an audible range.

The Autocratic *abs* Function

When you are only interested in positive numbers, use `abs` (absolute function). It has the syntax shown in Figure 5-6. Feed it a negative number and it gives back the same number without the negative sign: a positive number. Feed it a positive number and the number is unaffected. Thus, the `abs` function is used when you want to know the difference between two variables but don't care if the difference is positive or negative.

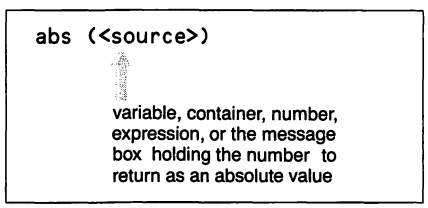


Figure 5-6. The `abs` function syntax.

The absolute value comes in handy in HyperTalk because it provides a way to make sure the output of calculations is "safe." For example, if you want to display a graph in HyperCard, you must make sure there are no negative numbers fed to the paint tools, and `abs` is used to guarantee this.

Example 17:
Enter `abs(-100)` in the Message box and press the Return key.

Result:
The number 100 appears in the Message box.

Function Arguments: More Things to Care About

So far, our functions have been simple. Feed the function a value, a variable, or a container, and it delivers a result. Functions are actually more supple than that and can be fed more than one value. To bring such a function to life, type its name followed by the multiple values you wish it to process. The *place holders* for these multiple values are called the *parameters* or *arguments* of the function. The HyperTalk syntax for functions requires that these arguments be separated by commas. Fur-

ther, each argument can be a value, variable, container, or expression. Only certain functions in HyperTalk require multiple arguments. For example, the `average` function, which we study next, accepts a list of values. There are other functions, such as `compound`, that require two arguments. When you learn how to make your own functions, you see that multiple parameters are very useful.

average: Just Another Pretty Function

The `average` function is used when you wish to know the average of a list of values. The `average` function is useful for calculating an average of sales or a three-month running average of a stock, projection estimates, statistics, scientific data, and even for games. The syntax for `average` is shown in Figure 5-7.

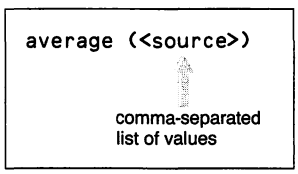


Figure 5-7. The `average` function syntax.

Example 18:

Enter `average(1,2,3,4,5)` in the Message box and press Enter.

Result:

The number 3 appears in the Message box.

Although you don't see it, the `average` function first adds all the numbers in the list (1 + 2 + 3 + 4 + 5) to make 15. It then divides this by the number of items in the list (5), giving the result of 15/5, or 3 and returning it to the Message box.

We do not include spaces between the comma-separated list in Example 18. HyperCard does not care if you have spaces between the commas or between the function name and the parentheses, so you can get away with statements like

```
abs  (    -111     )
```

but we scowl at such misuse of spacing in this book and in your scripts as well. Strive to make your style readable and clear as a bell.

Advanced Scripting: Romancing the Calculator

Let's expand the calculator to enable it to make random numbers and generate a set of average numbers. Don't worry about the numberFormat and Large buttons. They'll be covered later in this chapter.

Example 19:

1. Go to your HyperTalk Bible stack and open the Calculator card.
2. Add the following buttons and scripts to it and arrange them so they look like Figure 5-4.

```
on mouseDown -- random button
  put the random of card field x into card field Result
end mouseDown

on mouseStillDown -- average button
  put random(card field x) & "," after card field Result
  put average(card field Result) into Message box
end mouseStillDown

on mouseUp -- clear button
  put empty into card field Result
  put 1 into card field x
  put 1 into card field y
end mouseUp
```

3. Enter the number 25 in the top field, x, and press the random button repeatedly.

Result:

Each time you click the random button, it generates a random number between 0 and the number in the top register.

4. Press the clear button. It puts the value 1 into the two top registers. Increment the top register to 10 or so, using the arrow buttons, and press and hold down the mouse button over the average button for a few seconds.

Result:

Random numbers are generated in the Result card field, separated by commas. When you release the mouse button, the average of these numbers is calculated and placed in the Message box. The average of the random numbers should be close to one-half the number in the top field of the calculator.

Evening It out: *round*

Numbers you wish to use as whole numbers often have fractional or decimal remainders. The value of these decimal remainders can be important. For example, you may have a list of decimal numbers to convert into whole numbers but you wish to keep them as close as possible to the original decimal numbers. The `round()` function takes a decimal number and converts it to the nearest whole number, rounding the decimal part up or down. The syntax is shown in Figure 5-8.

Let's see how `round` works in the Message box:

Example 20:

1. Go to your HyperTalk Bible stack and open the card containing the "Poorman's Pseudoscientific Calculator."
2. Enter `round(1.49)` in the Message box and press the Return key.

```
round (<source>)
```

value, container,
expression, or function

Figure 5-8. The `round()` function syntax.

Result:

The value 1 appears in the Message box showing that `round` rounded .49 down to 0.

3. Enter `round(1.5)` in the Message box and press the Return key.

Result:

The value 2 appears in the Message box, showing that `round` rounded the .5 up to 1.

The rule appears to be that if the decimal part of the number is .5 or greater, the number is rounded up to the next highest whole number and if the decimal part of the number is less than .5, it's rounded down to the next lowest whole number. However, there is more to `round` than meets the eye. Try this:

Example 21:

1. Enter `round(1.5)` in the Message box and press the Return key.

Result:

HyperTalk displays 2 in the Message box. That makes sense.

2. Enter `round(2.5)` in the Message box and press the Return key.

Result:

HyperTalk displays 2 again in the Message box. That doesn't follow the rule!

There is an addition to the rounding rule. The `round` function changes its actions depending on whether the whole number is even or odd. If the whole number is odd, `round` rounds `.5` up. If the whole number is even, `round` rounds `.5` down. This is a good idea because, if you always round the value `.5` downward, or always round it upward, you favor one direction over another. Thus, if you are averaging lots of values, you tend to get lower values.

Dropping the Whole Thing: *trunc*

HyperTalk has another interesting command that works somewhat like `round`, but is different. The `trunc` function truncates a decimal number by just stripping off any fractional portion. Its syntax is shown in Figure 5-9.

Example 22:

1. Enter `trunc(1.49)` in the Message box and press the Return key.

```
trunc (<source>)
```

value, container, expression, or function returning a decimal number

Figure 5-9. The `trunc()` function syntax.

Result:

1 appears in the Message box.

2. Enter `trunc(1.99)` in the Message box and press the Return key.

Result:

1 appears in the Message box.

The decimal part is dropped. Try other values like 0, – 12.111, and .000001.

INT = trunc

BASIC hackers have to build their own rounding function using BASIC's `INT()` function. `INT()` works like HyperTalk's `trunc()`, that is, it simply drops the decimal part of the number. In BASIC, you must write `LET X = INT(X + .5)` to round off the number in `X`. If you plug different values into `X`, this works just as writing `round(x)` does in HyperTalk.

Precedence

When you mix simple arithmetic operations with functions, you often want to change the order in which the calculations are made. Let's see why. Suppose you must add two numbers and apply a tax to the sum to get a total. You might do it like this:

```
put item1 + item2 * taxrate into total
```

HyperTalk does not complain about the results, but the answer is probably not what you want because HyperTalk does not add `item1` and `item2` first. Rather, it performs the multiplication between `taxrate` and `item2`, and, when done, adds `item1` to this total. This decision to do the multiplication before the addition is not arbitrary. There is a fixed set of rules for how operators and functions are evaluated. One operator or function has precedence over another. This set of rules is expressed in the HyperTalk Precedence Table, which explains the priority evaluation for the operators and functions. (See Table 5-2.)

Operators with higher priorities operate before those with lower priorities, but if they are equal in precedence, they're evaluated from left to right. For example, in the equation

Table 5-2. HyperTalk Precedence Table

Order	Operators	Operations
1	( )	Natural precedence override
2	– , not	Negative and logical inversion
3	^	Exponentiation
4	*, /, div, mod	Multiplication, division, modulus
5	+, –	Addition and subtraction
6	&, &&	Concatenation
7	>, <, >=, <=, contains, is in	Logical greater and less than
8	=, <>	Logical equal and not equal
9	and	Logical and
10	or	Logical or

```
10^2 * 3
```

exponentiation is evaluated before multiplication. First, 10 is raised to the second power (100) and then the result is multiplied by 3 (300).

If you have a calculation where there are equal priorities, such as

```
9 div 2 * 2
```

the results are evaluated left to right, so the example gives 8 as a result.

What if you really are interested in 9 divided by the product of 2 times 2? You need to let HyperTalk know you want to change the normal precedence rules. Do this by using parentheses. For example, if you rewrite the equation like this:

```
9 div (2 * 2)
```

the parentheses force HyperTalk to first evaluate the product of 2 times 2, then take that result and do the `div` with 9, giving the result of 2.

Obviously, parentheses are critical to the health of good math scripts. There are some operators in Table 5-2 that we have not covered. These are explained in chapters 10 "Repeats" and 11, "Strings."

When functions are mixed with operators, the functions are always evaluated by HyperTalk first, acting somewhat like parentheses. In the equation

```
2^abs(4) + 10
```

the number 2 is first raised to the 4th power (16) and then 10 is added to it (26). However, if we write

```
2^((abs(4) + 10))
```

the result is 2 raised to the 14th power (16,384), quite a different value.

The Large and Small of HyperTalk Numbers

We saved the best part for last. You may have been wondering what the largest and smallest numbers are that can be entered into HyperTalk. You may have wondered how accurate the HyperTalk environment is and how many digits of precision it offers. If you plan to do any serious number crunching—if you have calculations to perform and want the results to be accurate—you'd better suffer along with us on this.

It's in SANE

Playing with large numbers using HyperCard's Message box as a calculator brings interesting results. Try this:

Example 23:

1. Enter `put pi^26` in the Message box and press the Return key.

Result:

The value `8431341691876.207044` appears in the Message box.
There are 19 digits in this number. Remember 19. It's an important number.

2. Enter `pi^100` into the Message box and press the Return key.

Result:

The huge number

`518784831431961313800000000000000000000000000000000`

appears in the Message box.

There are 19 nonzero digits out of 72 total digits.

The Apple Macintosh is built around a special mathematics library called SANE (Standard Apple Numerics Environment). This library is stored in the ROM of the Macintosh and in part of the larger system file on disk. The SANE library treats numbers differently than a scientific calculator.

There are 72 digits in the result of step 2 of Example 23 because that is the maximum number of digits a SANE number can have. The SANE package only allows up to 19 of these digits to hold actual significant values. The other digits are zeros. These 19 significant digits are called the *digits of precision*. They set the overall accuracy for HyperTalk calculations. SANE (and HyperCard) keeps track of the decimal point internally, so numbers larger or smaller than 19 digits are padded with zeros when they are displayed.

Engineers and scientists (and some calculators and other computer languages) prefer to represent large numbers in a special condensed format called scientific or *exponential notation (e-notation)*. There is a precision part and a decimal multiplier, which eliminate having to write large numbers of zeros for large numbers. For example, `1.234567 E06` means the number 1.234567 with 6 decimal places to the right of the decimal point: 1.234567 million, or 1,234,567. HyperTalk, however, does not use e-notation and, instead, sees numbers in terms of a maximum of 72 digits. If the

number has more than 19 digits in it and a calculation is performed, HyperTalk adds more zeros, up to a maximum of 72 digits.

Why No Scientific Notation?

If you are doing scientific number-crunching you may be pulling out your hair at this point. Many on-line debates reveal tempers flaring over the lack of e-notation in HyperCard. Proponents of e-notation argue that it allows large numbers to be entered with few keystrokes, thereby increasing the probability of accurate results and making the cruncher's job easier. Opponents of e-notation claim that neither cash registers, watch calculators, mileage computers, or digital tuners can understand scientific notation, so why put it into HyperCard, which is supposed to be a universally understandable tool? After all, they claim, how many HyperCard users want to know "the elasticity of steel in gigawhatsits."

Keep in mind that HyperCard's creators at Apple had information management, not number-crunching, in mind when they made HyperCard. Because HyperTalk is an interpreted language (see the Introduction), it is not suitable for intensive number-crunching. No doubt, there are numerous applications wherein scientific notation would be desirable, but with the philosophy of HyperCard being to function in the database world, it is likely to receive improvements in these areas before we see advances in the area of number-crunching. There is, however, the probability that some smart person will write an XCMD that adds e-notation to all the HyperTalk numbers. See chapter 13, "Math Functions," for a more detailed discussion of scientific notation and a routine that creates exponential notation in HyperTalk.

Advanced Scripting: Big Numbers in Scripts

Let's see how these big numbers look in HyperTalk by writing a script that puts the largest HyperTalk number into a card field. We also see how complicated it is to put large numbers into scripts.

Example 24:

1. Go to your HyperTalk Bible stack and open the Calculator card.
2. Create a new button, name it `Large`, and arrange the button after the clear button as shown in Figure 5-4.
3. Put the following handler into it:

```
on mouseUp
  --       >    >    >    >  <- these are comments to help typing
  put "999999999999999999990" into card field x
  put "00000000000000000000" after card field x
  put "00000000000000000000" after card field x
  put "0000000000000" after card field x
  --       >    >
  put card field x * 1 into card field Result
end mouseUp
```

4. Click the mouse on the Large button. It fills the top field of the calculator with the largest number HyperTalk can handle using SANE.

Result:

The number appears as

```
999999999999999999900000000000000000000000000000000000000000000000000000000
1...5....1....1....2....2....3....3....4....4....5....5....6....6....7...
```

which is 19 digits of 9 followed by another 54 digits of zeros, for a total of 73 digits. (Although SANE claims to hold up to 72 digits, our empirical results show up to 73 are possible.)

The last statement in the script, `put card field x * 1 into card field Result`, multiplies this number by `1`, so your original value is preserved. To see that this is the largest number, use the calculator as follows:

Example 25:

Type `1.1` into the second calculator field, y. With the largest number still in the upper register, press the exponentiation button to raise the largest number to the power 1.1.

Result:

? appears in the Result box, showing that the number overflowed HyperTalk's SANE package.

When you do HyperTalk calculations and the results are displayed in fields or the Message box, make sure your fields are capable of storing the full 72 digits. If the field is too small, it only displays the portion that shows in the justification set for the field. That may be okay, but be aware of it.

If the number exceeds 72 digits and you do any calculations on it, the amazingly profound ? suddenly appears rather than your desired number. ? is an error message meaning the number is too large to be represented by SANE. This may be a reason for storing intermediate results in a temporary variable and examining it before it is displayed to see whether it has ? in it. If you find ? in the variable rather than a legal number, give the user a warning that the number is too large to display.

Masking the Result: *numberFormat*

Remember, when HyperTalk performs calculations, it keeps track of them internally with up to 19 digits of precision and allows a number with up to 72 total digits to the right or left of the decimal point. Externally, however, HyperTalk presents numbers to you in a *filtered format*.

Example 26:

Enter `1/3` in the Message box and press the Return key.

Result:

`0.333333` appears in the Message box.

Here, we divide 1 by 3. There are only six 3s to the right of the decimal point. If there are actually 19 digits of precision maintained internally, shouldn't there be 19 3s to the right of the decimal point in this particular calculation rather than just six?

Right! We see only six decimal digits and one 0 because, in HyperTalk, the internal representation of numbers is filtered through a property called `numberFormat` before the numbers are displayed. This property controls how the internally stored results of calculations appear on the screen, in the Message box, or in containers with respect to the decimal point. The `numberFormat` property is controlled by the string you type. The characters of this string control the way digits are displayed according to a certain format. Figure 5-10 shows the syntax of the `numberFormat` property. Figure 5-11 shows how you visualize the way the property filters the internal representation.

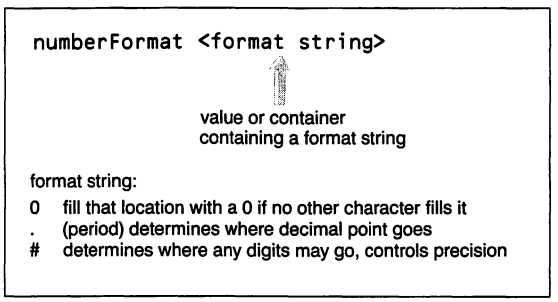


Figure 5-10. The `numberFormat` property syntax.

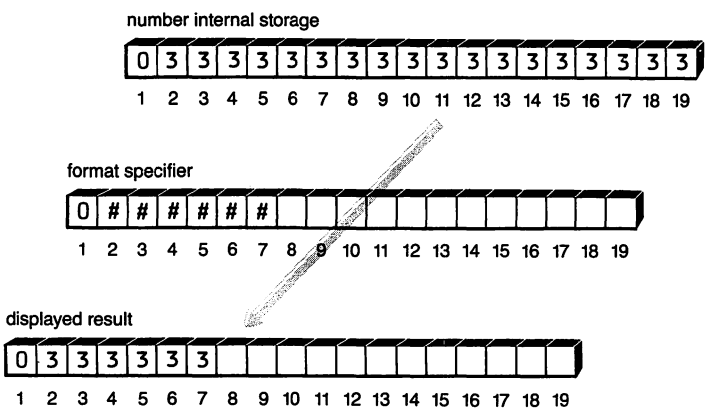


Figure 5-11. `numberFormat` acts like a filter.

The `numberFormat` property acts like a gate to the internally stored digits. The

72 digits of the internal representation of the number are adjusted so that different digits on either side of the decimal point are displayed.

When HyperTalk starts, it sets the digits of `numberFormat` to display up to 6 digits to the right of the decimal point and a single zero to the left of the decimal point if there is no whole number. (Otherwise, it displays the whole number itself.) This state of affairs can be changed, however. See how the `numberFormat` property is specified by the following example:

Example 27:

Enter `the numberformat` in the Message box and press the Return key.

Result:

`0.######` appears in the Message box.

Because `numberFormat` is a global property, our example causes the current value in the `numberFormat` string to be displayed in the Message box. This is the default HyperCard has set up. How is the string of characters interpreted? As shown in Figure 5-10, the three characters in the format string have special meaning as filters. A zero `0` character on either side of the decimal point tells HyperTalk to place a zero in the final displayed digit of any number if there is no corresponding digit in the initial number to be displayed. For example, if you want to display dollars and cents in your fields, your format string might be `0.00`. If the number to be displayed is 7.3, HyperTalk would fill in the digits as `7.30`, so it looks like dollars and cents.

The crosshatch character `#` tells HyperTalk to display a digit in that location in the final number (but not a zero if there is not a digit in that location). The previous example shows that the standard default for `numberFormat` is six digits to the right of the decimal and a zero to the left in the units digit. That is why we get results like `0.333333` when we type `1/3` in Example 26. Here is an example showing how the `numberFormat` string works in a script:

Example 28:

1. Create a new field, name it `Format`, give it a shadow type, set its font to 9-point Monaco, and arrange as shown in Figure 5-4.
2. Create a new button, call it `numberFormat`, and arrange it as shown in Figure 5-4.
3. Enter the following script into the numberFormat button:

```
on mouseUp
  put "#.####" & return into card field format
  set numberFormat to card field format
  put pi * 1 & return & return after card field format
  --     <---9--->
  put "#.#########" & return after card field format
  set numberFormat to last line of card field format
  put pi * 1 & return & return after card field format
  --     <---------22--------->
  put "#.#######################" & return after card field format
  set numberFormat to last line of card field format
```

continued

```
  put pi * 1 & return & return after card field format
  --     <------------26---------->
  put "0.00000000000000000000000000" & return after card field
format
  set numberFormat to last line of card field format
  put pi * 1 & return after card field format
end mouseUp
```

4. Click the numberFormat button.

Result:

The field under the button displays four format strings and the resulting numbers obtained after the `numberFormat` property is set using these strings.

Note that in the third section in step three of this example (`put "#.#######################" & return after card field format`), when we use more `#` characters than there are places in the internal representation of `pi`, the rest of the places are not filled in. In the fourth section, where we used `0` for the symbols instead of `#`, the digits not in the internal representation are padded with zeros.

Use `numberFormat` when you want to make a group of numbers line up in columns. By setting the number format with the `0` character, you are sure that every output number has the same number of digits displayed.

There is one subtle weirdness about `numberFormat`. Format changes affect only the results of subsequent arithmetic operations. Thus, if you just change the format in a script, it does not affect current values. If a number is displayed in a field, changing `numberFormat` does not affect it until some operation is performed on that number.

If you set the `numberFormat` from the Message box, it only works for the next operation and after that reverts back to the default mask. In fact, whenever the `idle` message is received, `numberFormat` automatically reverts back to the `0.######` default, so you must use `numberFormat` in scripts.

Constants Are Strings, Not Numbers

The constants, such as `ten` or `pi`, actually represent strings of digits rather than a number. Only when used in a calculation by a math operator does HyperTalk actually convert the strings of digits to real numbers. Internally, HyperTalk represents the resulting calculation at its highest precision, but if you display the number in a container, such as the Message box or a field, the internal number is filtered by the `numberFormat` setting, and so may look different. For example, typing `put pi` in the Message box results in `3.14159265358979323846`, the string of digits representing `pi`. If you type `put pi +0` in the Message box, you get `3.141593`. `pi` is converted to a number because of the `+` sign. This number is then converted back to a string when sent to the Message box because the Message box is a container. The default number format rounds `pi` to six decimal places.

Project Script: Chaos Theory and Verhulst

An interesting new idea called *chaos theory* attempts to find order and pattern in what seems to be simple randomness or erratic behavior. Chaos deals with such subtle and eerie effects as demonstrating the effect of a butterfly's wings on the weather or mathematically generating mountain ranges in computers. Chaos theory predicts the position of a leaf as it floats down a river, and shows how the temperature of melting snow affects a river's tumbles and turns. Unlike the classical theory of cause and effect, chaos theory says that phenomena might be controlled by *zones of stability* and *instability*, and these zones can be measured and examined. One way to investigate chaos theory is to examine population. When Malthus predicted in 1798 that population always increases until stopped by a shortage of food, biologists searched for an equation that describes the drop in a population when it exceeds a fixed food supply. One formulation that comes close is Verhulst's Law of 1845, which is expressed in the equation

$$x_{next} = rx(1-x)$$

Here, x is the population and varies from 0 = extinction to 1 = maximum. The factor r is the rate of growth, which is a constant in the formula.* Start with any value of x between 0 and 1, and then iterate through successive generations of x. Each time you calculate an x, use it in the next calculation. Amazingly, there are magic values of r that change how the values of x behave. Although most of the numbers generated by the formula start by swinging up and down, they quickly settle to one stable population (x = .6552). This is called the *stabilization value*. Roughly speaking, the waves generated by the equation move up and down for awhile, decrease in amplitude, and eventually settle down to that single value, x = .6552. We are more interested, however, in values of r that cause instability in the waves. In the final example of this chapter, we use everything learned so far to create a script that generates motions and sounds for populations following different Verhulst numbers:

Example 29:

1. Go to the HyperTalk Bible stack, create a new card, and name it `Chaos Theory`.
2. Create a new button, name it `chaos`, place it roughly as shown in Figure 5-12, and enter the following script in it:

```
on mouseStillDown -- button chaos
  global x, r, y
  set numberFormat to ".########"
  put r * x * (1 - x) into x
  put x into card field Population
  put trunc(180 - x * 100) into y -- this is just to move the hand
  get the loc of card button Plot
```

continued

*The functions controlling and limiting population growth are now believed to be more complex than Malthus realized. For a complete discussion of this equation, see James Gleick's *Chaos: Making a New Science* (New York: Viking Penguin Inc., 1987), especially page 70.

```
  put y into item 2 of it
  set the loc of card button Plot to it
  play "harpsichord" tempo 500 60 + trunc (y / 20)
end mouseStillDown
```

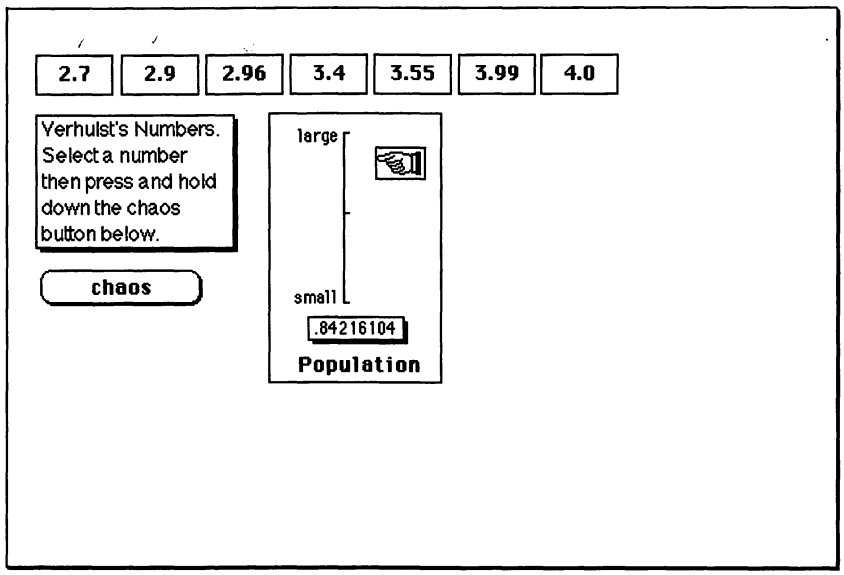


Figure 5-12. Chaos Theory card.

3. Now create a new button, set its name to `2.7` and its type to rectangle. Enter the following script in it:

```
on mouseUp -- buttons 2.7, 2.9, 2.99, 3.4, 3.57, 4.0
  global r, x
  put the short name of the target into r
  put .01 into x
  put r into msg
end mouseUp
```

4. Make five copies of the 2.7 button, rename them `2.9`, `2.99`, `3.4`, `3.57`, and `4.0` and place them as shown in Figure 5-12.
5. Make a transparent button, name it `Plot`, and turn its `show name` property off. Give the button the left-pointing hand icon and size and locate it as shown in Figure 5-12. This hand represents the fluctuating population.
6. Select the line tool and draw a vertical line in the paint layer of the card as shown in Figure 5-12. Select the text tool and add the labels next to the vertical line as shown. Surround the whole thing with a rectangle and label it `Population`. This area displays the changing population.
7. Create a new field and name it `Population`. Make it a shadow rectangle type of field and locate it as shown in Figure 5-12. This displays the changing values of the population equation.

8. Click a number button. That sends a Verhulst number to the chaos formula. Hold down the mouse over the chaos button.

Result:

You see and hear some very interesting results.

The first three numbers cause the equation to swing up and down and produce a wide variable in sound. As the formula continues to operate, the numbers begin to swing between two values. Eventually, these two values become one. That way, three numbers cause different rates of convergence between values in the formula, but eventually all settle down to a steady tone. As the numbers in the population box show, the number settles down to around `.65526014`. (We use `numberFormat` to increase the visible digits of precision to 8.)

In chaos theory, these Verhulst numbers represent stable populations, because the system eventually settles down to a steady state condition and a single value. These are not what we are interested in, however. As you increase the value of r in the equation, something strange happens. A value of `3.4` sets the population into what eventually turns out to be stable oscillating state, and the hand swings up and down between two values. Eventually, these values settle down to a single stable number and the equation produces a stable swing between two different values. The next two values of r also give stable swings but take longer to settle down. Finally, the value of `4.0` represents complete chaos—the hand jerks up and down and the music sounds as if it's from outer space. This is a zone of instability or chaos.

Researchers in chaos theory are most interested in the critical boundary between steadiness and oscillation. If you carefully control the r values sent to the chaos button in the regions from stable oscillation to chaos, you find certain critical points where the population numbers split in two again and oscillate between several stable values. These points are called *bifurcations* and they follow a fascinating pattern of splitting faster and faster, until they suddenly break off and go into pure chaos. There appears to be instability and randomness but, when you look closer, you see great amounts of stability and order. That is the heart of chaos theory—order within randomness.

The scripts in the Verhulst buttons are all the same:

```
on mouseUp
  global r, x
  put the short name of the target into r
  put .01 into x
  put r into msg
end mouseUp
```

This script sets the value of `r` and the initial value of `x` for the calculation made in the chaos button. The variables `r` and `x` are declared *global*. Chapter 4, "Containers and Values," demonstrated that a global variable can be read by any handler that also declares the variable global. This means our chaos button is aware of the variables `r` and `x` when it starts, and these variables will thus assume the value that is set by this button. The statement `put the short name of the target into r` is a clever way to get the name of the button into a variable. Remember, `the target` is the object that

causes the running handler to execute. Because the user clicks the button, `the target` becomes the button that is clicked. Set `x` to `.01` and put `r` into the Message box so you can verify it is the right value.

The code for the chaos button looks like this:

```
on mouseStillDown
  global x, r
  put r * x * (1 - x) into x
  put x into msg
  put trunc(x * 100 + 100) into y -- this is just to move the hand
  get the loc of card button Plot
  put y into item 2 of it
  set the loc of card button Plot to it
  play "harpsichord" tempo 500 60 + trunc (y / 20)
end mouseStillDown
```

This script uses a `mouseStillDown` trap so the handler repeats while you hold down the mouse button. We talk about `mouseStillDown` in detail in chapter 6, "Messages." The variables `x` and `r` are declared global. The values of `r` and `x` have been set. The formula is calculated, and the result is used to set the vertical location of the hand icon. The value sets the note played in harpsichord voice.

Summary

By now, you should have a pretty good idea of how to perform math in HyperTalk. You have seen how to work with the math operators and how to do running totals with the math commands. You are now familiar with the HyperTalk constants and know `pi` from `zero`. You learned the `INF` and `NAN` error messages, and the overflow error—the question mark. We showed you the operators `div`, `mod`, and `^`. You learned what functions are and how to use some of them, including `random`, `average`, `round`, and `trunc`. You also learned to control HyperTalk's numbers display with `numberFormat` and how the order of precedence affects calculations. You built a Poorman's calculator and explored the precision of HyperTalk. We showed you how Apple's SANE math package works, and you learned the limits of the largest number HyperTalk can handle. Finally, we presented a project script that taught you to appreciate chaos theory.

Review Questions

1. Match the correct HyperTalk operator, command, or function in column one with its corresponding purpose in column two.

a. `+`	e. `by`
b. `^`	f. `div`
c. `add`	g. `mod`
d. `/`	h. `round`

i. `multiply` command keyword
j. remainder operator
k. integer division operator
l. raise-to-a-power operator
m. `round` function
n. `add` command
o. divide operator
p. add operator

2. The variable `note` holds 1.5, `base` holds 50, and `rate` holds 200. Write commands to do the following:

   a. raise `note` to the 2nd power and add it to `base`.
   b. raise the integer of `base` times `note` to the power of `pi`.
   c. subtract the rounded value of `note` divided by 3 from card field foo and make foo a running total.
   d. find the average of three random numbers, each calculated with a maximum that is `random(pi)`.

3. Are the parentheses optional in the answer to question 2a?
4. Predict or calculate the results that appear in the Message box for the following calculations:

   a. `2^3`
   b. `pi^200`
   c. `random(10000000000)`
   d. `random of 6 mod pi`

5. How would you format a number so that it prints in dollar-and-cents format (with a dollar sign to the left of the largest digit) and fills any nondigit space with a 0? Assume the number you want to print is x and you want to display it in the Message box.
6. Will the answer to question 5 work if the `numberFormat` global property is set from the Message box? If not, why not?
7. What do the following calculations evaluate to in HyperTalk?

   a. `round(1.5)`
   b. `round(2.5)`

8. How do you get a variable in a button to cycle from 0 to 60 for as long as the button is pressed?
9. Describe the range of numbers generated by these statements:

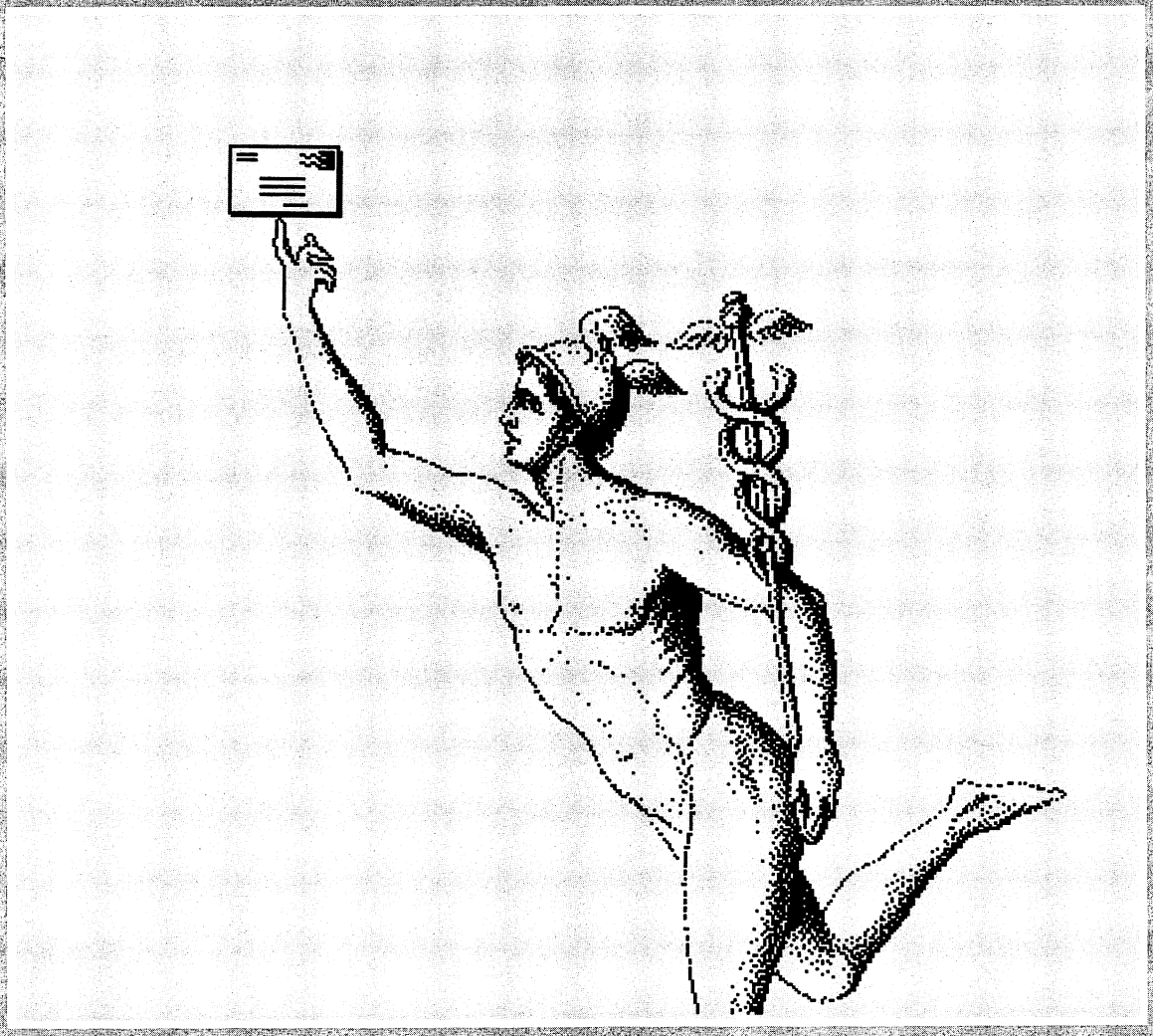
   a. `average(random(100),random(200),random(300))`
   b. `put round(the random of 100 / the random of 5)`

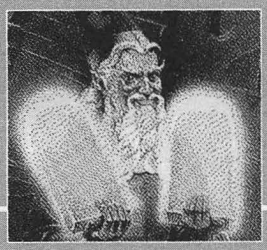
Exercises

1. Create a button that makes random music as follows: Make a button and name it `Random Music`. Write a button script that plays a random note as long as the button is held down. Create two other buttons that change the tempo from 500 and from 1000.
2. Copy the Random Music button, rename it `Sweep`, and write a script for it that cycles though 60 notes and plays that sequence repeatedly, without overflowing HyperTalk.

Chapter 6

Messages





Quick View

Keywords

pass
send
help

Messages

mouseLeave
mouseEnter
mouseWithin
startUp
idle
suspend
resume
quit
returnInField
enterInField
newField
newField
closeKey
enterKey
returnKey
tabkey
doMenu

Concepts

Messages
Parameters
Reporting messages
Mouse messages
Within and leave
Handler sharing
Button messages
Messages to fields
Unlocking fields with command key
Keyboard messages
Tabbing in fields
Trapping function, control, arrow keys
Birth and death of object messages
Combining messages
Passing messages
Command messages
Menu commands
Changing the message hierarchy
Handler libraries

Chapter 6

Messages

Messages are the life and breath of HyperCard. Without messages, nothing happens; the various objects sit there, holding their little scripts and waiting in vain for an inspiration. Looked at another way, messages are the glue that holds HyperCard objects together. In a word, messages are important.

In this chapter, we fill in your knowledge of messages by adding to what you learned previously. In fact we take another pass at the whole subject of messages, commands, handlers, objects, and object hierarchy and add discussions of such related subjects as parameters, the `pass` statement, and the `send` command. This chapter shows where messages enter the hierarchy, where they can be intercepted, how you can create your own messages, and how you can redirect the message hierarchy for your own purposes.

A Quick Review

Recall that HyperCard objects (buttons, fields, cards, backgrounds, and stacks) float in a sea of messages. These messages are constantly being sent from HyperCard, the mouse, the keyboard, the objects themselves, and from scripts. The messages enter the object hierarchy at the field and button or the card level and then are funneled up through the background, stack, and Home stack objects until they reach HyperCard. Figure 6-1 shows the basic object hierarchy with all the message sources indicated.

You might think of messages as radio broadcast signals that are received by HyperCard objects. The signals flood the screen, but we never actually see them.

Note that some sources send a broad signal to fields and buttons as well as to cards, whereas others broadcast only to cards. At least, that is what happens if the messages are not intercepted en route by a message handler located in one of the object scripts. Figure 6-2A shows what happens to a message when it is not "handled," and Figure 6-2B shows what happens when it is handled.

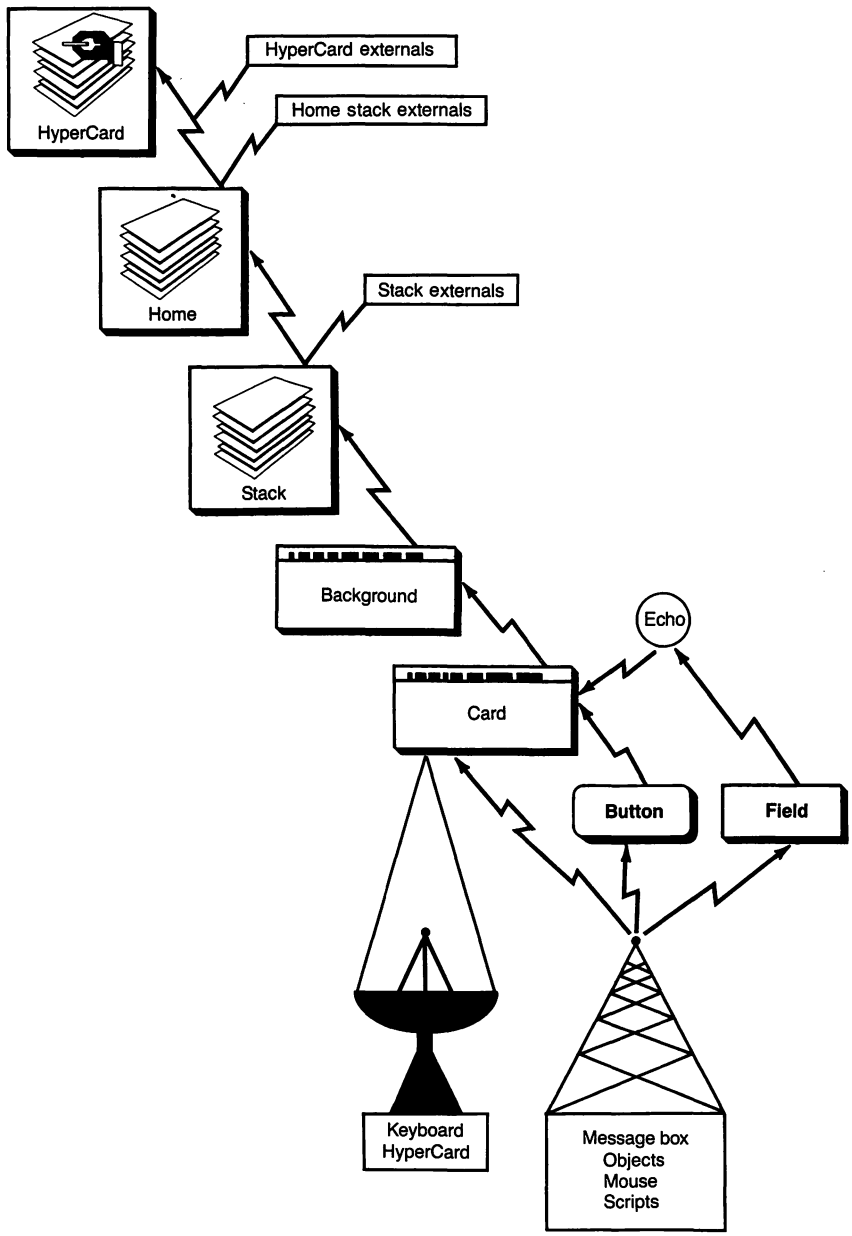
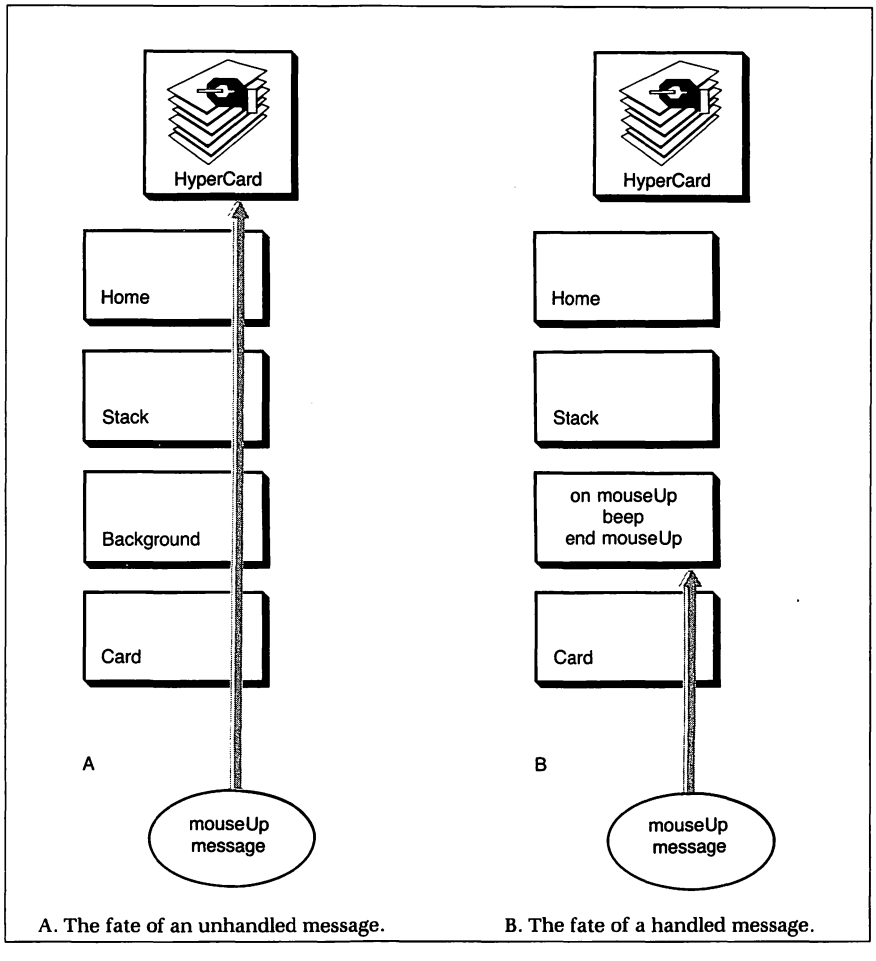


Figure 6-1. Object hierarchy and the sources of messages.

Note that a handled message is used up or absorbed and goes no further up the hierarchy (though you see some exceptions to this later in the chapter).



A. The fate of an unhandled message. B. The fate of a handled message.

Figure 6-2. The fate of unhandled and handled messages.

Introducing Parameters

Not all messages are as simple as the `mouseUp` message we've demonstrated in earlier chapters. The `mouseUp` message is a good example of what we call *reporting messages*. These messages simply report that some event has occurred, in this case, that the mouse button has been released after being pressed. Sometimes you need more information from a message than what is contained in the message name itself. For instance, you might need to know which of the four possible arrow keys has been pressed. This information is called an *argument* and is passed along with the message in the form of a single parameter. Although a few reporting messages, like `arrowKey`, pass a parameter, most parameters are passed by command messages. Remember that all HyperCard menu items, HyperTalk commands, and functions

work by sending messages up the object hierarchy to HyperCard. Examples are `go home` or `answer "Alert!"`. In these cases, `home` is the argument passed with the `go` command message telling where to go, and `"Alert!"` is the parameter passed with `answer` telling what to display in the answer dialog box.

With a little work, you can retrieve these parameters in a handler that then reinterprets the command so that it does whatever you want it to do. Although the functions used to retrieve command message parameters aren't covered until chapter 14, "Advanced Commands and Functions," in the following section on keyboard messages, we show you a simple retrieval technique to access the parameters passed by our reporting messages. Please note that parameters, despite what you hear in the conversations of people trying to sound important, have nothing to do with limits or boundaries.

Tale-Bearing Messages

Reporting messages are like Paul Revere: They go from object to object with the news that something has happened to see whether any object wants to do something about it. If an object contains a handler to trap that message, the handler is run and the message is absorbed. (You see an exception to this at the end of the chapter.) If there is no handler, the message keeps going up the hierarchy of objects until it reaches HyperCard itself, where it stops. (If it is a command message, it is handled by the application HyperCard, causing the activity associated with the command.) Reporting messages can be subdivided into four groups on the basis of where they are generated. The groups are: mouse messages, keyboard messages, object messages, and HyperCard messages, as shown in Table 6-1.

Table 6-1. HyperTalk Reporting Messages

Type	Reporting Messages
Mouse	mouseDown, mouseStillDown, mouseUp, mouseEnter, mouseWithin, mouseLeave
Keyboard	returnKey. enterKey, tabKey, controlKey, functionKey
Object	newStack, newBackground, newCard, newButton, newField, deleteStack, deleteBackground, deleteCard, deleteButton, deleteField, openStack, openBackground, openCard, openField, closeStack, closeBackground, closeCard, closeField
HyperCard	startUp, suspend, resume, idle, help, quit

Messages from Your Mouse

Mouse messages pertain to either the coordinate position of the mouse cursor on the screen or the up/down status of the mouse button. The mouse screen position messages report when the mouse cursor (actually the hot spot at the fingertip of the browsing cursor) crosses into, crosses out of, or lies within the rectangle containing

a field or button. These messages only enter button or field objects. Although these messages do not enter at the card level directly, they are passed to the card and on up the hierarchy if they are not handled at their field or button entry point. The screen position messages sent when the mouse cursor passes over a button or field are `mouseEnter`, `mouseLeave`, and `mouseWithin`. The `mouseEnter` message is sent at the instant the hot spot of the cursor crosses into a button or field. As long as the cursor remains in the button or field, a stream of `mouseWithin` messages is sent, alternating with `idle` messages. Finally, the instant the hot spot of the cursor crosses out of the field or button, a `mouseLeave` message is sent. In the first example, you write handlers for two of these three messages.

Example 1:

1. Open the last card of your HyperTalk Bible stack, add a new background, and name the first card of the new background `Messages`.
2. Make a new card button and name it `Coming & Going`.
3. Enter the following handlers in the Coming & Going button script:

```
on mouseEnter -- Coming & Going button
  put "Welcome O noble rodent" into message box
end mouseEnter

on mouseLeave -- Coming & Going button
  put "Farewell O noble rodent" into message box
end mouseLeave
```

4. Move the cursor in and out of the Coming & Going button.

Result:

Whenever the cursor enters the button, the mouse is greeted. When the cursor leaves, the mouse is bid farewell.

Our message handlers sit in the button script and wait for their message to come along. When the cursor passes over the Coming & Going button, a `mouseEnter` message is immediately sent from the mouse to the button. Because the button contains a handler for the message, the on mouseEnter handler runs, displaying `Welcome O noble rodent!` in the Message box. When the cursor moves out of the button, the `mouseLeave` message is sent, and the on mouseLeave handler runs, displaying `Farewell O noble rodent!` in the Message box.

Now, add a `mouseWithin` message handler to this button to make the example more interesting.

Example 2:

1. Open the Messages card of your HyperTalk Bible stack.
2. Add the following handler to the Coming & Going button script:

```
on mouseWithin -- Coming & Going button
  put "Eeek!" into message box
  wait 30 ticks
  put "A mouse!" into message box
  wait 30 ticks
end mouseWithin
```

3. Change the mouseEnter handler as follows:

```
on mouseEnter -- Coming & Going button
  put "Welcome O noble rodent" into message box
  wait 30 ticks -- new line, lets you see message
end mouseEnter
```

4. Move the cursor in and out of the Coming & Going button.

Result:

Between the greeting and the farewell, the Message box alternates between displaying `Eeek!` and `A mouse!`.

These three messages (`mouseEnter`, `mouseLeave`, and `mouseWithin`) are only sent when the cursor is over a button or field. Although they are not sent directly to higher objects, like cards or backgrounds, once a button or field receives a message, it passes it up if the message doesn't find a handler in the button or field. In the next example, you cut the handlers from the button and paste them into the card script to demonstrate that these messages pass up the hierarchy if they are not handled.

Example 3:

1. Open the Messages card of your HyperTalk Bible stack.
2. Cut the three handlers in the Coming & Going button by selecting the text and pressing Command-X.
3. Open the card script and paste the handlers in with Command-V.
4. Move the cursor in and out of the Coming & Going button.

Result:

As before, the cursor is greeted when it enters the button and is bid farewell when it leaves. Between the greeting and the farewell, the Message box alternates between displaying `Eeek!` and `A mouse!`.

5. Add another card button to the card, reselect the browse tool, and move the cursor in and out of both buttons.

Result:

Exactly the same results as before for both buttons.

What we have here is a case of handler sharing. Because the three handlers were moved from the button to the card level, instead of applying only to the one button, they now apply to any button (or field) that doesn't have its own handlers.

We talk more about sharing handlers in a later section, but note that it doesn't stop there. Move the same handlers into the background script and they respond whenever a cursor is in a button in any card belonging to the background. Move them to the stack, and their scope is still wider (assuming more than one background to a stack). Move the same handlers into the stack script of your Home stack, and you see the same results for every button and field in any stack you open. Figure 6-3 illustrates the principle of sharing handlers by position. Figure 6-3A shows the button and field handlers not being shared, Figure 6-3B shows the card handler being shared with the objects below it (note the fill pattern), Figure 6-3C shows the background handler being shared by a wider group of objects, Figure 6-3D shows the stack handler being shared with a still wider group, and in Figure 6-3E, the

Home stack handler fills in the gaps but is capable of being shared with all the objects in the figure.

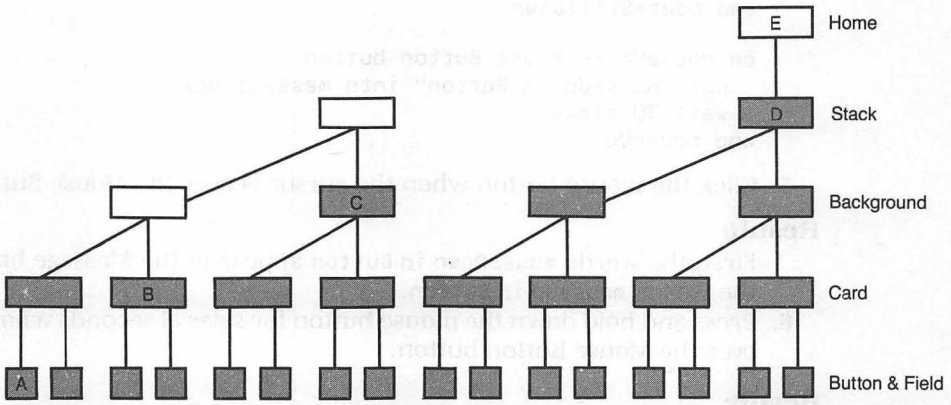


Figure 6-3. Sharing handlers by position.

Mouse Button Messages

The other three mouse reporting messages apply to the status of the mouse button rather than to the location of the mouse cursor on the screen. When you press the mouse button, a `mouseDown` message is sent to whatever object is below the mouse cursor at that instant. If the cursor is above a button, the button gets the message. If the cursor is above a field and the field is locked, the field gets the message. If there are neither buttons nor fields under the cursor, the card gets the message. If the card doesn't have a handler for the `mouseDown` message, the message goes up to the card's background and then to the stack the card is a part of. (Keep in mind that although on the screen messages go down to the button and then deeper to the card, the background, and the stack, they are really going up the hierarchy.)

Example 4:

1. Open the Messages card of your HyperTalk Bible stack.
2. Cut the Example 3 handlers from the card script and put them back into the Coming & Going button.
3. Name the button you made in the last example `Mouse Button`.
4. Enter the following handlers in the Mouse Button script:

```
on mouseDown -- Mouse Button button
  put "mouseDown in Button" into message box
  wait 30 ticks
end mouseDown
```

continued

```
on mouseStillDown -- Mouse Button button
  put "mouseStillDown" into message box
  wait 30 ticks
  put "in Button" into message box
  wait 30 ticks
end mouseStillDown

on mouseUp -- Mouse Button button
  put "mouseUp in Button" into message box
  wait 30 ticks
end mouseUp
```

5. Click the mouse button when the cursor is over the Mouse Button button.

Result:

First, the words `mouseDown in Button` appear in the Message box, followed by the words `mouseUp in Button`.

6. Press and hold down the mouse button for several seconds when the cursor is over the Mouse Button button.

Result:

First, the words `mouseDown in Button` appear in the Message box, followed by alternating `mouseStillDown` and `in button` messages until the mouse button is released and the words `mouseUp in Button` appear.

7. Press the mouse button when the cursor is over the Mouse Button button, and don't release it until the cursor is outside the button.

Result:

Same `mouseDown in Button`, `mouseStillDown`, and `in button` messages appear, but `mouseUp in button` is not displayed because the button does not get the `mouseUp` message.

This is how all Macintosh buttons are supposed to work. Pressing the mouse button causes an event that should trigger some visual feedback to let the user know that releasing the mouse button activates the button, but if you slide off the screen button, without releasing the mouse button, the screen button should not be activated.

By writing handlers for these six messages, you can make your buttons perform different tasks in different situations. You can change a button's name or its icon, move it, play a sound, or show an explanatory note in a separate field whenever the mouse cursor passes over a button with a mouseEnter handler. With a mouseWithin handler, you can make a button invert back and forth so that it appears to flash or play a tune, or you can do animation by cycling through icons. When the cursor leaves a button, you can trap the `mouseLeave` message to restore the screen and the button to the way it was before the cursor entered, or you can display a message thanking the user for visiting. Imagine a whole stack of overly polite buttons that thank you on all messages . . . the mind boggles.

Use the `mouseDown` message to warn the user that something is about to happen—something like the function of the musical score in a suspenseful movie. The `mouseUp` message is used constantly in HyperCard to change radio buttons and check box buttons or to perform whatever action is appropriate for a given button.

Messages to Fields

Buttons aren't the only objects that can receive messages from mice. Fields can receive all six of the messages we covered in the last section. We say *can* because a normal field interprets the user's pressing the mouse button as an instruction to place the insertion point cursor on the line under the hand cursor. This is a normal thing for a text window or field to do, but it prevents `mouseDown` and `mouseUp` messages from getting to handlers in the field. Recall, however, that we said *normal.* There are two ways to bypass the field's capability for confiscating messages.

First, you can hold down the Command key as you click. The I-beam cursor changes to a hand cursor when the Command key is pressed. If you then click the mouse button, the messages get through. Of course, you don't want to do this all the time, and you don't want to have to explain this to everyone who uses your stacks.

The second way to get a message through is to open up the field info dialog box for the field and check Lock text (or you can set this property in a script with the statement, `set the lockText of field <name> to true`). Either of these techniques lets `mouseDown` and `mouseUp` messages pass through to your handlers. The `mouseEnter`, `mouseWithin`, and `mouseLeave` messages get through regardless of the `lockText` value.

We demonstrate the use of mouse messages in fields and of the text editor bypass techniques mentioned previously in this next example.

Example 5:

1. Open the Messages card of your HyperTalk Bible stack.
2. Make a new card field, set its style to rectangle, name it `Mouse-to-field`, and type `Mouse-to-field` into the field.
3. Copy the three handlers from the Coming & Going card button and paste them into the field's script.
4. Copy the three handlers from the Mouse Button card button and paste them into the field's script. Alter these handlers as follows:

```
on mouseDown -- Mouse-in-field field
  -- replace "Button" with "Field"
  put "mouseDown in Field" into message box
  wait 30 ticks
end mouseDown

on mouseStillDown -- Mouse-in-field field
  put "mouseStillDown" into message box
  wait 30 ticks
  -- replace "Button" with "Field"
  put "in Field" into message box
  wait 30 ticks

end mouseStillDown

on mouseUp -- Mouse-in-field field
  -- replace "Button" with "Field"
```

continued

```
    put "mouseUp in Field" into message box
    wait 30 ticks
end mouseUp
```

5. Move the cursor in and out of the Mouse-in-field field without pressing the mouse button.

Result:

The cursor changes to an I-beam when it is over the field, but as before, the cursor is greeted when it enters the button and is bid farewell when it leaves. Between the greeting and the farewell, the Message box alternates between displaying `Eeek!` and `A mouse!`.

6. Press and release the mouse button when the cursor is over the field.

Result:

The flashing bar of the insertion point is positioned on the line under the cursor, indicating that the field is ready to receive text.

7. Move the I beam cursor outside the field to get the browsing cursor back, press and hold the Command key, move the cursor back over the text in the field, and then press and release the mouse button.

Result:

The cursor shape stays a hand, the cursor is greeted by `Welcome o noble rodent` in the Message box, and then that greeting is replaced with alternating displays of `Eeek!` and `A mouse!` until the mouse button is pressed. The display then changes to `mouseDown in Field`, followed by alternating `mouseStillDown` and `in field` displays. When the mouse button is released, `mouseUp in field` is displayed before returning to the `Eeek!` and `A mouse!` displays. When the cursor finally leaves the field, there is a final `Farewell O noble rodent` display.

8. Try step 5 again, but this time press the mouse in a part of the field without any text. (Curiously, the `mouseStillDown` message does not register, and you see a `mouseUp in field` display even though the mouse button is still down. Bug or feature? You be the judge.)
9. Select the Field Tool from the Tool menu, select the field by clicking it, select Field Info in the Objects menu, check the Lock text box, close the dialog box, and select the browse tool again.
10. Press and release the mouse button when the cursor is over the field.

Result:

A reprise of the results of step 5, but this time `mouseStillDown` and `mouseUp` work correctly even when the cursor is over a blank line of text.

There are several interesting things to note here. An unlocked field interprets the `mouseEnter`, `mouseWithin`, and `mouseLeave` messages the same way a button would, but `mouseDowns` are interpreted in a way that makes sense for text fields. (`mouseStillDowns` and `mouseUps` are used for dragging a selection.) However, if you really want to handle these messages in a field, you can use the Command key technique, or you can lock the text. Of course, you can't type text into a locked field, so it is protected from casual editing, but you can use a locked field for everything you can use a button for. Keep this in mind if you ever need a wordy button.

With the mouse reporting messages covered in this section, you can create scripts that interact with the mouse. Various things can be made to happen depend-

ing on whether the cursor is entering, inside of, or leaving an object. Likewise, your scripts can react differently to the mouse button being pressed down than to its being let up. When you add the information provided by the mouse functions covered in the next chapter, you have the tools to make your scripts respond to the mouse in almost every imaginable way.

Messages from Your Keyboard

If the users are not using the mouse, they usually interact with the computer through the keyboard. Chapter 8, "Keyboard," is devoted to this subject, but we give you a "messages perspective" of the keyboard here. You already know how to type text into fields and into the Message box, but you will also want to use keys rather than screen buttons to do repetitive tasks, such as moving from field to field or from card to card.

Not every key can send a HyperTalk message. In fact, on the original 128/512KB Macintosh keyboard, only three keys have this capability: the Tab, Return, and Enter keys. The entry point for all the keyboard messages is the card level. The messages then work their way up the hierarchy from there. In the next example, we demonstrate handling the three keyboard messages available to everyone.

Example 6:

1. Open the Messages card of your HyperTalk Bible stack.
2. Add the following handlers to the card script:

```
on enterKey
  dial 1
end enterKey

on returnKey
  dial 5
end returnKey

on tabKey
  dial 9
end tabKey
```

3. Press the Tab key, the Return key, and then the Enter key.

Result:

Pressing each of the three keys causes a different dial sound.

Unreliable Keyboard Messages

These keyboard messages are not as reliable as the mouse messages because when the browse tool is being used to edit a field and when the text tool is being used to paint text on the screen, the Return and Enter key messages are not sent in Version 1.0.1 of HyperCard. This is an issue you need to be aware of before you plan a script, but it doesn't mean these messages can't be used to good effect.

returnInField and *enterInField*

Two new messages, introduced in HyperCard version 1.2, address the uncertainty of `returnKey` and `enterKey` messages being sent when a text field is active. If you press the Return key when the insertion point cursor is blinking in a field, a `returnInField` message is sent to the card. Pressing the Enter key in the same situation sends an `enterInField` message. With these messages, use the Return key as an alternative tabbing device and the Enter key to move to the next card, to send some data to a file, or to do whatever you please.

Messages from Bigger Keyboards

Moving up to the Macintosh Plus keyboard, we can add the four arrow or cursor keys to the list of message-sending keys. The standard Eastwood keyboard for the SE and Mac II adds a Control key, and the extended keyboard adds 15 function keys. These 20 keys pass parameters with their messages. We won't give you all the tools for extracting and evaluating the parameter values until later in the book, but be assured that when an `arrowKey` message is sent, the name of the actual key pressed is sent as well and can be extracted by a script. The same is true with the `controlKey` message, which includes the number of the key that was pressed at the same time as the Control key, and the `functionKey` message, which passes the number of the function key pressed. For now, we just include a script that identifies which arrow key was pressed. If your keyboard doesn't have cursor keys, skip this example.

Example 7:

1. Open the Messages card of your HyperTalk Bible stack.
2. Add the following handler to the card script:

```
on arrowKey which -- Messages card
  put which into message box
  dial 3
end arrowKey
```

3. Press the cursor keys.

Result:

Pressing any of the cursor keys causes the same dial sound but displays a different parameter value (up, down, left, or right) in the Message box.

In this example, we used the variable `which` to hold the parameter value, but any other variable name works just as well (`var`, `parameter`, `key`, etc.). As you see in the Message box, when the up cursor key is pressed, the `arrowKey` message passes the value up. When the right cursor key is pressed, the right value is passed, and so on. If you have a keyboard with a Control key or with function keys, you can substitute `controlKey` or `functionKey` in Example 7 for `arrowKey` and use the revised handler to read the parameters passed by those messages as well.

Example 8:

1. Open the Messages card of your HyperTalk Bible stack.
2. Add a new card button and name it 1. Make a note of its ID number (we refer to it as button id 21).
3. Enter the following handlers in button ID 21:

```
button id 21:on mouseWithin
  set name of me to the short name of me + 1
end mouseWithin

on mouseStillDown
  set name of me to the short name of me - 1
end mouseStillDown
```

4. Enter the following handlers in the card script and deactivate the existing enterKey and returnKey handlers by commenting out their first lines:

```
on enterKey -- card script
  set name of button id 21 to 100
end enterKey

on returnKey -- card script
  set name of button id 21 to "1"
end returnKey

on tabKey -- card script
  set name of button id 21 to short name of button id 21 + 10
end tabKey
```

5. Move the cursor over the button and watch the name change.
6. Press the mouse button over the field.

Result:

When the cursor is over the button, the number in the button slowly increments. When the mouse button is pressed, the number quickly decrements.

7. Press the Tab key several times.

Result:

The number in the button goes up 10 for each press of the Tab key.

8. Press the Enter key.

Result:

The number changes to 100.

9. Press the Return key.

Result:

The number goes back to 1.

Note the difference in the rate the number changes when it is going up and when it is going down. This is a result of the `mouseWithin` message alternating with the `idle` message, which cuts in half the number of messages that get to the mouseWithin handler in a set period of time. If you switch statements in those two handlers, the number then gets bigger quickly and gets smaller slowly.

Keyboard messages give people with a Control key or function keys the ability to customize their keyboards. Pressing a function key by itself or pressing the Control

key and another key can start a handler that does anything you want it to do. These keys can substitute for buttons in stacks where screen space is at a premium or where you want to provide options that you don't want to be obvious to the browser.

Messages from Objects

One of the features of an object-oriented programming language like HyperTalk is that the objects are smart enough to volunteer information about themselves that can be used in your scripts. This information takes the form of messages that are sent during the principal programming-language-object life crises: birth, death, etc. When an object is created, it sends a `new...` message (`newButton`, `newStack`, etc.). When it is cleared or deleted, it sends a `delete...` message (`deleteField`, `deleteBackground`, etc.). There are also messages for opening and closing an object (except for buttons). See Table 6-1 for a complete list of these messages.

Buttons and fields send these messages to themselves first, and if they are not handled there, they go up the object hierarchy. Cards, backgrounds, and stacks send their messages to the card first. From the card, a message passes up the hierarchy. What this means is that you can put a script into an object that will react to the message concerning itself. For instance, you may want to compact a stack whenever it is closed, or you may want to protect the information in a stack by having the stack react to being deleted by making a copy of itself. You may want to show related information when a field is opened and hide that information when the field is closed.

Although this last suggestion, using the `closeField` message, normally works, there are times when the `closeField` message is not sent, for instance, when the content of the field is unchanged or when another field is scrolled. Also, `openField` and `closeField` messages are not sent to a locked field. Consequently, you should always provide an alternative method for hiding the related information.

Example 9:

1. Open the Messages card of your HyperTalk Bible stack.
2. Disable the keyboard handlers in the card script by typing comment hyphens in front of the first line of each handler.
3. Add a new card field and name it `Field Messages1`.
4. Enter the following handlers in the field's script:

```
on newField -- Field Messages1 field
  play "boing"
end newField

on openField -- Field Messages1 field
  beep
  put "openField" into message box
end openField

on closeField -- Field Messages1 field
  dial "4"
  put "closeField" into message box
```

continued

```
  wait 30
end closeField

on deleteField -- Field Messages1 field
  dial "1234"
end deleteField
```

5. Select the field and then clone a copy by holding down the Option key while you drag the field with the mouse.

Result:

A boing sounds when the clone is set down—the newField handler at work.

6. Name the cloned field `Field Messages2`. Select the browse tool and click Field Messages1.

Result:

A beep sounds and openField appears in the Message box.

As you can see, thanks to our handlers, we now get audio feedback when a field is created or opened. Pressing the Tab key to go to another field is even noisier.

7. Type some random text into the field and then press the Tab key to move to the next field.

Result:

You hear a dial sound, `closeField` appears in the Message box followed by a beep, and then `openField` appears in the Message box.

8. Without changing the contents of the field, press the Tab key twice.

Result:

The tabs produce only a beep and an `openField` each time. (No `closeField` message is sent.)

9. Choose the Field tool from the Tool menu and then select and delete one of the two new fields.

Result:

Four dial tones followed by the deleting of the field.

As you can see, or hear, `closeField` really isn't very reliable, but we did pick up the field's death rattle thanks to the `deleteField` message. Note also that because the Mouse-in-field field was locked, it is not part of the tab cycle. Tabbing only opens unlocked fields.

Button objects can't be opened or closed, but the `newButton` and `deleteButton` messages work the same way the field messages work. The other objects generate and can handle all four object messages.

Example 10:

1. Open the Messages card of your HyperTalk Bible stack.
2. Cut the handlers from the remaining new card field and then paste them into the card script.
3. Modify the handlers as follows:

```
on newCard -- change all "Field"s to "Card" in the handlers
  play "boing"
end newCard
```

continued

```
on openCard -- Messages card
  beep
  put "openCard" into message box
end openCard

on closeCard -- Messages card
  dial "4"
  put "closeCard" into message box
  wait 30
end closeCard

on deleteCard -- Messages card
  dial "1234"
end deleteCard
```

4. Copy the card by selecting Copy Card in the Edit menu and then paste a copy by selecting Paste Card in the same menu.

Result:

A boing sounds when the copy is inserted.

5. Go back and forth between the two cards by pressing Command-2 and Command-3 or the right and left arrow keys.

Result:

Each time you switch cards, you hear a dial sound, `closeCard` appears in the Message box followed by a beep, and then `openCard` appears in the Message box.

6. Select Delete Card in the Edit menu.

Result:

Four dial tones are followed by the deleting of the card.

Combining Messages

We end our coverage of object messages with an example that demonstrates the variety of messages HyperCard sends when you do ordinary things.

Example 11:

1. If the TempStack created in chapter 4 is still around, open it. If it isn't, select New Stack from the File menu. Don't copy the background, and name the new stack TempStack.
2. Go to the Home stack and type the following handlers into the top of the stack script:

```
on deleteCard -- Home stack
  put ".deleteCard." after msg
end deleteCard
```

```
on deleteBackground -- Home stack
  put ".deleteBkgnd." after msg
end deleteBackground

on deleteStack -- Home stack
  put ".deleteStack." after msg
end deleteStack

on closeCard -- Home stack
  put ".closeCard." after msg
end closeCard

on closeBackground -- Home stack
  put ".closeBkgnd." after msg
end closeBackground

on closeStack -- Home stack
  put ".closeStack." after msg
end closeStack

on openCard -- Home stack
  put ".openCard." after msg
end openCard

on openBackground -- Home stack
  put ".openBkgnd." after msg
end openBackground

on openStack -- Home stack
  put ".openStack." after msg
end openStack
```

3. Go to TempStack.

Result:

The Message box shows all the messages sent during the transition from Home to TempStack (see Figure 6-4).

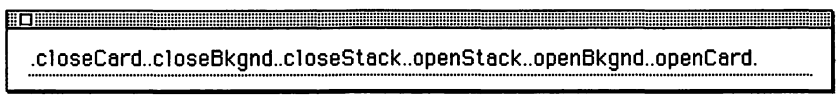


Figure 6-4. Messages sent when you go from one stack to another.

4. Clear the Message box and then select Delete Stack from the File menu.

Result:

The messages sent are displayed in the Message box (see Figure 6-5).

In Figure 6-5, we deleted the first message to see the last message.

You probably wouldn't have guessed that HyperCard would send the `deleteStack` message before the `close...` messages, but that's what it does. You can use the messages sent by any of the opening, closing, creating (new), and deleting events

```
.deleteStack..closeCard..closeBkgnd..closeStack..openStack..openBkgnd..openCard.
```

Figure 6-5. Messages sent when you delete a stack.

to activate handlers in any of the card and higher objects. You will probably use the `openCard` and `closeCard` messages most frequently, but `openStack`, `closeStack`, and `openField` are valuable as well. You probably will never need most of these messages, but if you should need to know when a background is deleted, the message is there just waiting to be handled.

startup and Friends

Other messages are sent by HyperCard itself. All these messages have their entry point at the card level. When HyperCard is launched, the first thing it does is send a `startUp` message to the card. If there is something you want to happen each time HyperCard starts, you can put it into a startUp handler in your Home stack. For example:

```
on startUp -- Home stack script
  answer "Welcome to HyperCard"
end startUp
```

There is already a startup handler in the Home stack's script that checks the Preferences card each time HyperCard is launched, but you could easily add this `answer` command to the end of that handler.

The *suspend* and *resume* Messages

Before HyperCard launches another application with the `open <application>` or `print... with <application>` commands, it sends a `suspend` message. When the launched application quits, program control returns to HyperCard and a `resume` message is sent. Sometimes HyperCard can't resume because something unusual happened while it was suspended (for example, the printer ran out of paper during a printing session), so you might want to write handlers to square away your stack when HyperCard is suspended and to return things to working order when the `resume` message is received. These handlers might look something like this:

```
on suspend -- stack script
  global anyVariable
  put anyVariable into card field "hidden" of first card
end suspend
```

continued

```
on resume -- stack script
  global anyVariable
  put card field "hidden" of first card into anyVariable
end resume
```

These two handlers save the value in variable `anyVariable` to a field (which is saved to disk) before the suspension and then restore the value to the global variable after HyperCard resumes operation.

The *idle* Message

Another message that HyperCard sends is the `idle` message. This message is sent when no other message is being sent. In fact, `idle` messages are even sent between `mouseWithin` messages. There are times when you will want to handle the `idle` message to make sure you don't miss something that doesn't generate its own message. The `idle` message handler is a last resort and should not be used often because it slows down your stack.

Example 12:

1. Open the Messages card of your HyperTalk Bible stack.
2. Add the following handler to the card script:

```
on idle
  beep
end idle
```

3. Move the cursor over the counting button (ID 21).

Result:

You hear a steady stream of beeps until you reach the button, and then you hear beeps alternating with the number incrementing.

4. Press the mouse button while the cursor is over the button.

Result:

The beeping stops as long as the mouse button is down and the number decrements.

This is an exercise to do quickly unless you are a glutton for beeps. In case you are wondering, the `idle` message seems to be sent every 4 ticks or every 1/15th of a second on a Macintosh with a 68000 microprocessor (Macintosh 128K to SE). Note that when the cursor is inside a field or button, `idle` messages and `mouseWithin` messages alternate. Pressing the mouse button cuts off both these messages and replaces them with `mouseStillDown` messages.

Recall that in chapter 4, you used an idle handler to display the current values of two global variables, `it`, and `the selection`. Once you learn about `if then` control structures and logical operators in chapter 9 "Ifs," you will be able to test for a wide range of conditions inside an idle handler. For example, you can find when the cursor is in a particular area of the screen, or you can compare the scroll value of two scrolling fields you want to always scroll together.

The *help* Message

HyperCard also sends a `help` message after you select the Help command from the Go menu or after you press the command key and the ? key. You can intercept this message before it gets to HyperCard to operate your own `help` messages.

Example 13:

1. Open the Messages card of your HyperTalk Bible stack.
2. Enter the following handler in the card script:

```
on help
  put "Consult the HyperTalk Bible" into message box
end help
```

3. Select Help from the Go menu.

Result:

`Consult the HyperTalk Bible` appears in the Message box.

The *quit* Message

The `quit` message is sent before HyperCard shuts down and the Finder takes over. Normally, you depend on the `closeStack` message to tell you when a session is over, but in cases where several stacks are used together and you are constantly going back and forth between them, you may need to use the `quit` message to signal that a session is really over and that it is time to store global variables. The best place to put a handler for the `quit` message is into the Home stack's stack script.

The *pass* Keyword

There are times when you want to intercept a message for your own purposes but still want the message to continue up the hierarchy. Sometimes you want to do this because you want to use the message twice, once in a low-level handler and again in a high-level handler. For example, you may have a mouseUp handler in a button that performs a task unique to that one button and another mouseUp handler in the card that does something else when any button on the card is pushed.

Other times you want to use the message, but you want HyperCard to get it also. This can be accomplished with a `pass` statement. Figure 6-6 shows the syntax for the `pass` keyword:

The next example demonstrates how this statement can be used to change the results of a previous example.

Example 14:

1. Open the Messages card of your HyperTalk Bible stack.
2. Edit the mouseDown handler in the Mouse Button button as follows:

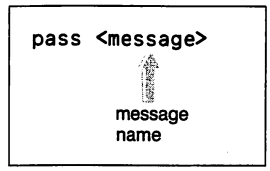


Figure 6-6. The pass keyword syntax.

```
on mouseDown -- Mouse Button button
  put "mouseDown in Button" into message box
  wait 30 ticks
  pass mouseDown -- new line
end mouseDown
```

3. Type the following handler into the card script.

```
on mouseDown -- Messages card
  put "mouseDown in CARD" into message box
  wait 30 ticks
end mouseDown
```

4. Press the mouse button when the cursor is over the Mouse Button button.

Result:

`mouseDown in Button` is displayed in the Message box and then replaced with `mouseDown in CARD`.

In Example 4, the `mouseDown` message was used up by the Mouse Button mouseDown handler, but now the message is passed on and handled a second time by the mouseDown handler in the card script. By adding another `pass mouseDown` statement to the card handler, the message could be handled yet again in the background script, and so on up to the Home stack. The `pass` statement allows you to write a general purpose handler at a relatively high level (card, background, stack, etc.) and still supplement that handler with special purpose handlers at a lower level.

Here's another example that uses the `pass` keyword so that you can add a little more value to a normal HyperCard feature.

Example 15:

1. Open the Messages card of your HyperTalk Bible stack.
2. Enter the following handler in the Messages card script:

```
on help
  put "Heeeers Carol"
  pass help
end help
```

3. Make sure the close and open handlers from Examples 10 and 11 are deactivated, then choose Help from the Go menu.

Result:

`Heeeers Carol` is displayed in the Message box, and then the HyperCard Help stack is opened (assuming it is available). To return to your HyperTalk Bible stack, press the ~ key.

Note that there are now two active help handlers in the Messages card script. HyperTalk always uses the first handler it comes to and ignores any others, so you don't have to deactivate a duplicate handler if it is below the one you want to run.

You also need the `pass` statement when you write handlers for the `doMenu` message later in this chapter. This is especially important because failing to pass on the `doMenu` message to HyperCard has the effect of disabling the Menu bar, which is very disconcerting.

Command Messages

We began this chapter by dividing messages into reporting messages (which report on events in the HyperCard environment) and command messages (which cause things to happen). Now that we have covered all the reporting messages, we can describe command messages, though we won't cover them all. Actually, it is impossible to cover all of them. Because you are not limited to the messages built into HyperTalk, you can make any number of your own commands and messages.

Although you may not have realized it, all HyperTalk commands (like `put`, `get`, `beep`, etc.) operate by sending messages to HyperCard. The point at which these messages enter the object hierarchy is the object containing the script that executed the command. In the case of menu commands and commands typed into the Message box, the message enters at the card object. By placing a handler into the path of these messages, you can intercept a command message just as you intercepted reporting messages. Let's begin by taking a look at some menu commands and their messages.

Menu Command Messages

Whenever you select a menu item in HyperCard, you send both a message and an argument along with it. The message is always named `doMenu`, whereas the argument is whatever the menu item is labeled. Therefore, when you select Help from the Go menu, as we did in Example 15, you pass as a parameter the argument `Help` along with the message `doMenu`. But wait! In Example 15, we intercepted a `help` message, not a `doMenu` message. Does the same message have two names?

The answer to that question is "no," but another example should make the situation clearer. In the next example, you write a handler for the `doMenu` message and see which handler gets what message.

Example 16:

1. Open the Messages card of your HyperTalk Bible stack.
2. Deactivate the help handler from Example 15, make sure the help handler from Example 13 is still present and active in the card script, and then enter the following handler in the card script:

```
on doMenu which -- Messages card
  put which into message box
  wait 30 -- ticks
  pass doMenu -- DON'T LEAVE THIS OUT OR YOUR MENU BAR WILL DIE
  show message window -- Safety measure
end doMenu
```

3. Select Help from the Go menu.

Result:

First the word `Help` is displayed in the Message box, and then it is replaced by `Consult the HyperTalk Bible.`

4. Select other menu items.

Result:

The name of each item appears in the Message box.

What is happening here? As shown in Figure 6-7, when you select the Help menu item, a `doMenu` message is sent to the card and then on up the hierarchy with the word `Help` as a parameter. When this message reaches the HyperCard level, HyperCard responds by sending out a `help` message to the card. Although this second message may seem redundant, it saves anyone wanting to intercept the `help` message (to provide custom help resources, for instance) the trouble of filtering the `doMenu` command parameter. Instead of playing with the `doMenu` parameter and worrying about passing on the message, you can write a handler, as we did in Examples 13 and 15, that waits for the simpler `help` message.

We included that `show message window` statement in the last example just in case you were foolish enough to ignore our advice and leave out the `pass doMenu` line. If you did, you can regain control of your Mac by entering `edit script of this card` in the Message box. This opens the script editor so that you can type the `pass doMenu` statement or comment out the doMenu handler.

The techniques for filtering the `doMenu` message (the parameter functions and the `if then` control structure) are explained in future chapters. For now, we just want you to know that even menu actions are driven by messages and that these messages can be handled just like any other message.

HyperTalk Command Messages

We certainly are not going to cover all the HyperTalk commands in this chapter, but we do want you to realize that commands in your scripts work by sending messages, which are actually copies of themselves, up the hierarchy to HyperCard. When you put the command `dial "345"` into a handler and that handler runs, it sends the message `dial`, which passes "345" as a parameter up to HyperCard—unless you intercept it first, of course.

Example 17:

1. Open the Messages card of your HyperTalk Bible stack.
2. Add a new card button to your HyperTalk Bible stack and name it `Dial`.

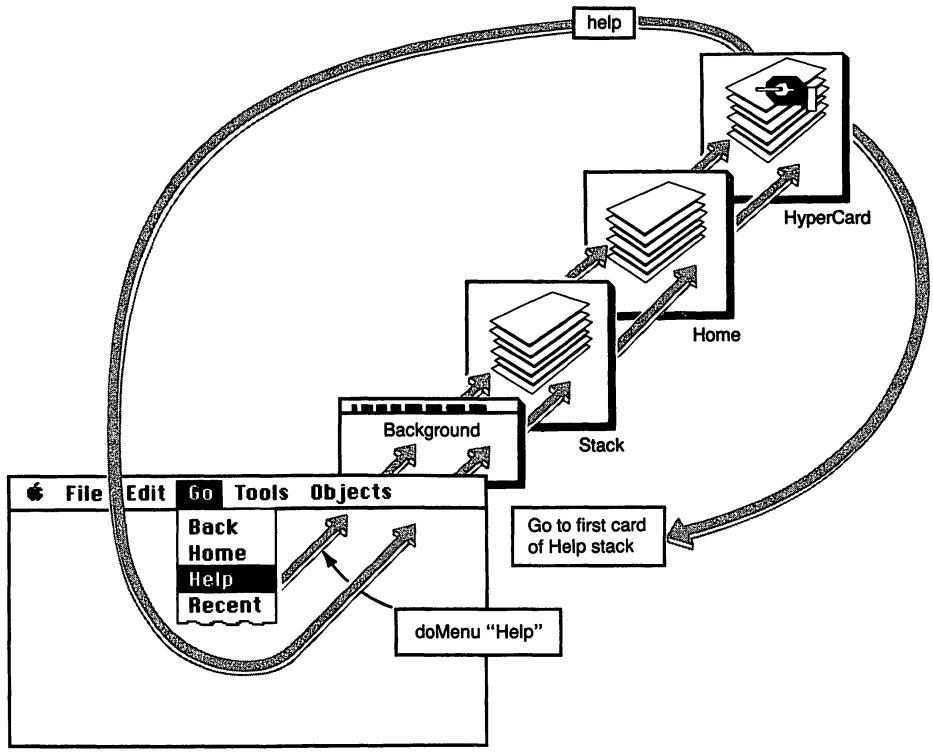


Figure 6-7. Messages sent by messages: the Help menu command in action.

3. Enter the following handlers in the Dial button's script:

```
on mouseUp -- Dial button
  dial "345"
end mouseUp

on dial -- Dial button
  play "boing"
end dial
```

4. Click the Dial button.

Result:
A boing sound.

The `dial` message didn't get out of the script that sent it because it was trapped by the dial handler. Although this is a trivial example, if you move the dial handler to the stack script, or even into your Home stack—it has the power to transform every `dial` command into a `play "boing"`. The same procedure works for any command in HyperTalk.

Bill Atkinson is fond of saying that HyperCard is a software erector set because

you can take the pieces and build anything you want. With HyperTalk, the flexibility is so great that if you aren't happy with the erector set pieces, you can reshape them any way you please. As if this weren't enough, HyperTalk also gives you the ability (as you will learn in the next section and in chapter 14, "Advanced Commands and Functions") to create your own unique messages and commands.

The *send* Keyword

Feeling subversive? The send keyword, the syntax of which is shown in Figure 6-8, breaks the usual button-field-card-background-stack-home hierarchy and lets you send messages directly to any object you choose and bypass other objects.

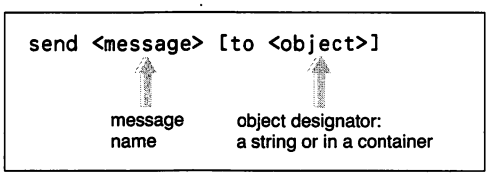


Figure 6-8. The send keyword syntax.

Suppose that you have a complex handler in a button script and that you want another button on the same card to do the same thing, plus other tasks as well. All you do is include the send command in the second button to send a message back to the first button. When the second button is pressed, it calls the first button with a send <message> statement. After the message handler in the first button has finished running, program flow returns to the line after the send keyword, and the script in the second handler resumes. Figure 6-9 shows how this works.

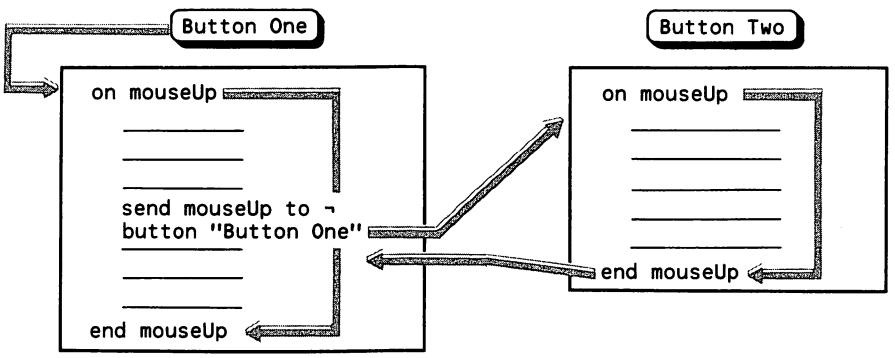


Figure 6-9. Program flow when a handler sends a message to another object.

***send* Equals Call a Subroutine**
For those of you with experience with other programming languages, `send` is HyperTalk's object-oriented equivalent to calling a procedure, a subroutine, or a subprogram. In HyperTalk, any handler in any object can become a subroutine for any other handler.

If you leave out the optional object designator parameter of the `send` keyword (the `to <object>` part), the message is sent first to the object containing the handler. This gives you an easy way to call other handlers in the same script. For example, `send help` sends a `help` message to the object containing the executing handler even if that object is a button or field. (Normally, `help` messages enter the hierarchy at the card level.) Sending a message without an object designator works the same way the `pass` statement works, except that program control returns to the sending handler after any handlers triggered by the `send` message finish.

The next example shows a very simple use of this command, but its implications are far reaching.

Example 18:

1. Open the Messages card of your HyperTalk Bible stack.
2. Add a new card button and name it `Send`.
3. Enter the following handler in the Send button:

```
on mouseUp -- Send button
  send mouseUp to card button "Dial"
  send mouseEnter to card button "Coming & Going"
  send deleteField to card field "Field Messages1"
  send help to card "Messages"
end mouseUp
```

4. Make sure the following handlers are in place:
   From Example 14

```
on mouseDown -- Messages card
  put "mouseDown in CARD" into message box
  wait 30 ticks
end mouseDown
```

   From Example 2

```
on mouseEnter -- Coming & Going button
  put "Welcome O noble rodent" into message box
  wait 30 ticks -- new line, lets you see message
end mouseEnter
```

   From Example 9

```
on deleteField -- Field Messages1 field
  dial "1234"
end deleteField
```

   From Example 13 (make sure the help handler from Example 15 is disabled

```
on help
  put "Consult the HyperTalk Bible" into message box
end help

on mouseUp -- Dial button
  dial "345"
end mouseUp

on dial -- Dial button
  play "boing"
end dial
```

5. Press the mouse button when the cursor is over the Send button.

Result:

First, `mouseDown in CARD` is displayed in the Message box prior to the mouse button's release. Then comes a boing from the Dial button, `Welcome O noble mouse` is displayed in the Message box, followed by four dial tones from the card field, and finally, `Consult the HyperTalk Bible` is displayed in the Message box.

You can even have handler chain reactions in which one button activates others that activate others, and so on. With this technique, you could easily use one button to simulate a user pressing any number of other buttons—an all-the-above button, if you will.

Zoom to Script with Shift Double Click
You can open the script of any object by holding down the Shift key and double-clicking the button or field object. You can do this from any of the tools.

Custom Messages and Handler Libraries

There is yet another subroutine possibility with the `send` keyword. So far, you have just been emulating the usual HyperCard and HyperTalk messages, but you are not limited to only these options. If you put the following handler into your stack:

```
on danWinkler
  put "Dan Winkler is the creator of HyperTalk" into the message box
end danWinkler
```

`danWinkler` can be used as a message, and `send danWinkler to <object>` sends a `danWinkler` message to the object holding this handler. The only limits on messages are that they have to have a handler or be an official HyperCard message.

This means that you can break down big handlers into smaller (subroutine) handlers. These smaller handlers are called from the main handler with the `send` command. If sections of several scripts are very similar, you can replace all the similar sections with one handler that only contains that section of script. Then, you

could call that section from all the other handlers that previously contained the similar lines. One of the great features of HyperTalk is that it encourages you to set up libraries of small scripts that you reuse over and over again. The possibilities are endless and only matched by what you'll find in chapter 14, "Advanced Commands and Functions."

Messages in the Message Box

The Message box is a two-way street. You can use it to display messages to the user or to send messages up the hierarchy. We've already used the Message box to send simple commands, but it can also be used with the `send` command to send reporting messages or, in fact, any messages, to a specified object. If no object is specified, the entry point for Message box messages is the card object.

Example 19:

1. Open the Messages card of your HyperTalk Bible stack.
2. Enter the following statement in the Message box:

```
help
```

Result:

`Consult the HyperTalk Bible` is displayed in the Message box.

3. Enter the following statement in the Message box:

```
send mouseUp to card button "Send"
```

Result:

As in Example 18, first you hear a boing from the Dial button, `Welcome O noble mouse` is displayed in the Message box, followed by four dial tones from the card field, and finally, `Consult the HyperTalk Bible` is displayed in the Message box.

As the help example shows, if a message has a handler in the card script or above, all you need to do to activate that handler is to type the message name into the Message box. If the handler is in a button or field, you must use the `send` command to direct the message to the proper destination.

Summary

In this chapter, you learned that HyperCard uses messages to communicate a wide variety of system events, including mouse, mouse button, keyboard, object, and HyperCard information, which we have referred to collectively as reporting messages. In addition, the menu selections and the commands in scripts or in the Message box all work by sending command messages up an object hierarchy, which can start with either buttons and fields or cards and progress up to the background,

stack, Home stack, and finally HyperCard levels. As you will learn in chapter 20, "Using XCMDs and XFCNs," these messages can go also to external resources before they reach HyperCard.

You learned that some reporting messages and almost all command messages pass arguments as parameters that can be accessed in their handlers. All these messages can be waylaid on their trip up the hierarchy by placing a message handler in their path. This handler can either use up the message or it can pass it on with the `pass` keyword. Handlers can also be activated by sending them a message from within another handler or from the Message box with the `send` keyword, which turns handlers into pseudo-subroutines. You've also seen how the meaning of a command can be changed by substituting a custom handler for HyperCard's usual action.

The `send` keyword gives you the option of using other handlers as subroutines and of creating your own messages. Handlers can also be shared by multiple objects by locating them higher in the object hierarchy so that one openCard handler can be used for multiple cards, multiple backgrounds, or even multiple stacks.

It takes time to get used to the options available in the HyperTalk language. Many of the topics discussed in this chapter are used every time you write a script, whereas you hardly ever use others. Some techniques you don't use at first, but they become standards later on. HyperTalk, an easy language to get started with, is far from simple. As you become more comfortable with messages and objects, you find more and more efficient ways to perform the same scripting tasks.

Review Questions

1. What do you call a hidden unit of information passed along with a message?
2. What stops a message from reaching HyperCard?
3. Which message is sent whenever the cursor crosses into a button or field?
4. What message alternates with `mouseWithin` when the cursor is inside a button or field and is also sent when no handler is running and no other message is being sent?
5. Name three keyboard messages that pass parameters.
6. When does pressing the Return key not send a `returnKey` message?

   a. when the browse tool is not selected
   b. when the insertion point cursor is blinking in a field
   c. when Jupiter aligns with Mars
   d. both a and b

7. What messages are sent when you click in a locked field?
8. What messages are sent when you click in an unlocked field?
9. What `open...` and `close...` messages are sent when you go from one card to another card in different backgrounds but in the same stack?
10. What message is sent only when the HyperCard application is launched?
11. What message is sent when HyperCard launches another application?
12. What message is sent when nothing is happening?

13. Which statement would you use if you want a `mouseUp` message handled in one object to continue up the hierarchy to another object or to HyperCard, and you don't want program flow to return?

    a. `pass mouseUp`
    b. `send mouseUp`
    c. `go away and stay away`

14. When is the `help` message sent?

    a. when the Help menu item is selected
    b. when the `doMenu "Help"` message is executed

15. Where does the `help` message enter the hierarchy?
16. Show the command statement you would use to direct a message, say `mouseUp`, against the hierarchy from a handler in a card script to a handler in the script for card button ID 3.
17. Where is the entry point for messages typed into the Message box?

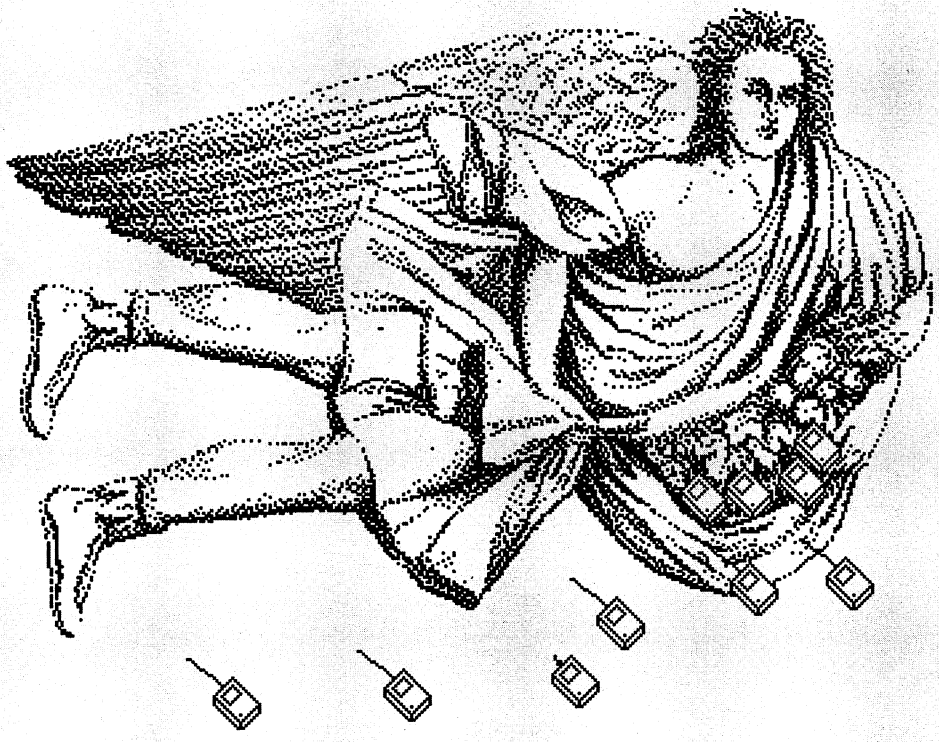
    a. the card
    b. the stack
    c. depends on where the Message box is
    d. 20525 Mariani Avenue

Exercises

1. Make two new card buttons that go to the next and previous cards, respectively, in response to a mouse click. Allow the user to choose a scroll or wipe visual effect and have the two buttons share the same music subroutine-handler to give some audio feedback.
2. Add a card script handler to give the user the option of going to either the next or the previous card when the mouse button is pressed over the card or over a button or field that doesn't have a mouseStillDown handler (hint). Use the handlers from Exercise 1 to actually go to the card requested.

Chapter 7

Mouse



Quick View

Commands
`click at`

Preview
The `wait` command

Keywords
`up`
`down`
`true`
`false`

Functions
`the mouseH`
`the mouseV`
`the mouseLoc`
`the mouse`
`the mouseClick`
`the clickLoc`
`the clickH`
`the clickV`

Concepts
The mouse state

Chapter 7

Mouse

The mouse is a vital part of the Macintosh personality, so, of course, HyperTalk has a host of mouse-related features. You've already seen the mouse messages and how they are used to initiate handlers. In this chapter, we describe several mouse functions, which return information about the mouse, and the mouse command `click at`, which lets a script simulate clicking the mouse.

The mouse functions most often are used in expressions that control the program flow. For example, you might instruct the Macintosh to take a certain action if the mouse button is down or to repeat playing a musical sequence until the mouse is clicked (that is, until the mouse button is pressed and released). Chapters 9, "Ifs," and 10, "Repeats," tell you how to express and employ such relationships, but before you can do that, you need to be able to obtain the raw data used in these relationships. That is, asking the question, "Has the mouse been clicked?" assumes you have some way of answering the question.

Fortunately, the HyperTalk mouse functions are specifically designed to answer these kinds of questions. We take a quick tour of these functions to see what questions they answer and how they work. For example, we see how to create a card so that its buttons are activated merely by having the mouse present in them. The next chapter covers similar keyboard functions. Then, we take up the real work of seeing how they can be used in scripts.

Where's the Mouse?

The Macintosh always keeps track of where the mouse is on the screen. (Keeping track of the physical mouse is your job, however.) The Macintosh uses the same coordinates as the `location of` property does; that is, it measures the position in pixels from the upper-left corner of the card. See Figure 7-1. Note that the coordinates are relative to the card, not to the screen. For small-screen Macintoshes, the card and screen normally coincide, so card coordinates and screen coordinates are the same. But, as you may remember from chapter 2, HyperCard normally centers the card on larger screens. In that case, it's important to know that the mouse position is relative to the card. Moving the mouse cursor to the left of the card makes the horizontal

position negative, and moving the cursor above the card produces a negative vertical coordinate.

For convenience, Hypertalk provides you with three functions for getting the mouse coordinates. `the mouseH` returns the mouse's horizontal position, `the mouseV` returns its vertical position, and `the mouseLoc` returns both.

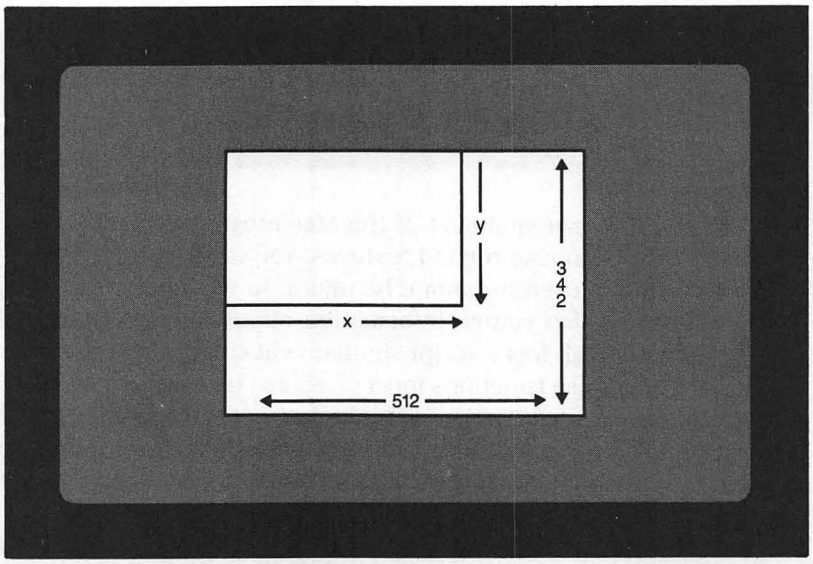


Figure 7-1. Mouse coordinates are relative to the card.

The *the mouseH* Function

The function `the mouseH`, the syntax of which is shown in Figure 7-2, returns the horizontal position of the mouse (its distance in pixels from the left side of the card). Often the term *x coordinate* is used for the horizontal position. The syntax is simple; unlike the math functions in chapter 5 "Simple Math," the mouse functions don't use arguments.

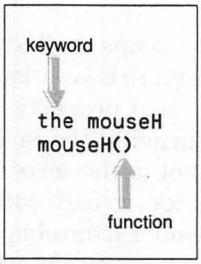


Figure 7-2. The `the mouseH` function syntax.

Note that you have your choice of two forms: a `the` prefix or a `()` suffix. This is true for all the mouse functions.

It's easy to check out how `the mouseH` works.

Example 1:

1. Go to your HyperTalk Bible stack.
2. Bring up the Message box and type `the mouseH`.
3. Press the Return key.

Result:

The position of the mouse, in pixels, from the left side of the card appears in the Message box.

4. Move the mouse horizontally, retype `the mouseH`, and press the Return key.

Result:

By gosh, the mouse has a new horizontal position, as shown in Figure 7-3.

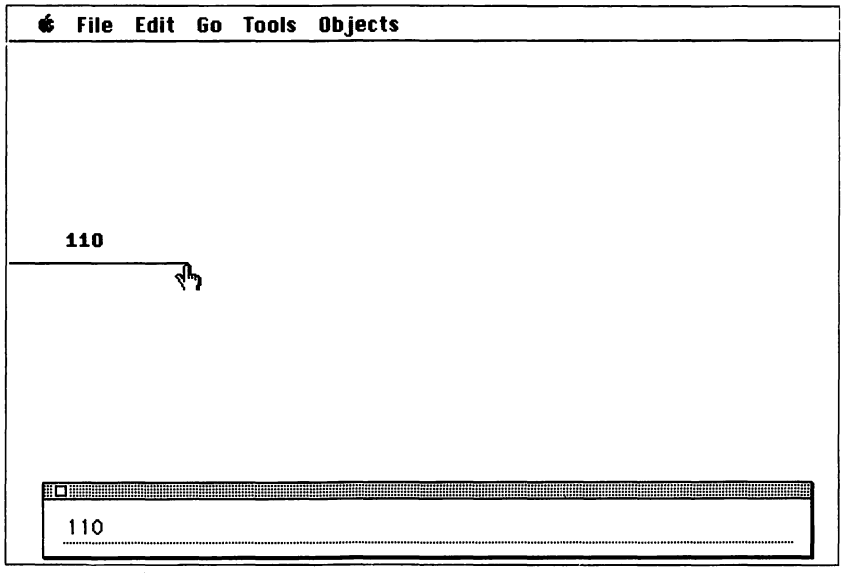


Figure 7-3. `the mouseH`

The *the mouseV* Function

The function `the mouseV` works just like `the mouseH` except that it returns the distance, in pixels, from the top of the card. Figure 7-4 shows its syntax.

Often the vertical position is termed the *y coordinate*. Unlike the usual y coordinates found in math textbooks, the Macintosh y coordinate increases downward rather than upward.

Example 2:

1. Go to your HyperTalk Bible stack.

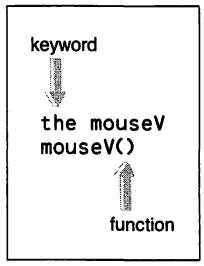


Figure 7-4. The `the mouseV` function syntax.

2. Bring up the Message box and type `the mouseV`.
3. Press the Return key.

Result:

The position of the mouse, in pixels, from the top of the card appears in the Message box.

4. Move the mouse vertically, and enter `the mouseV` in the Message box again.

Result:

The Message box reports a new location, as shown in Figure 7-5.

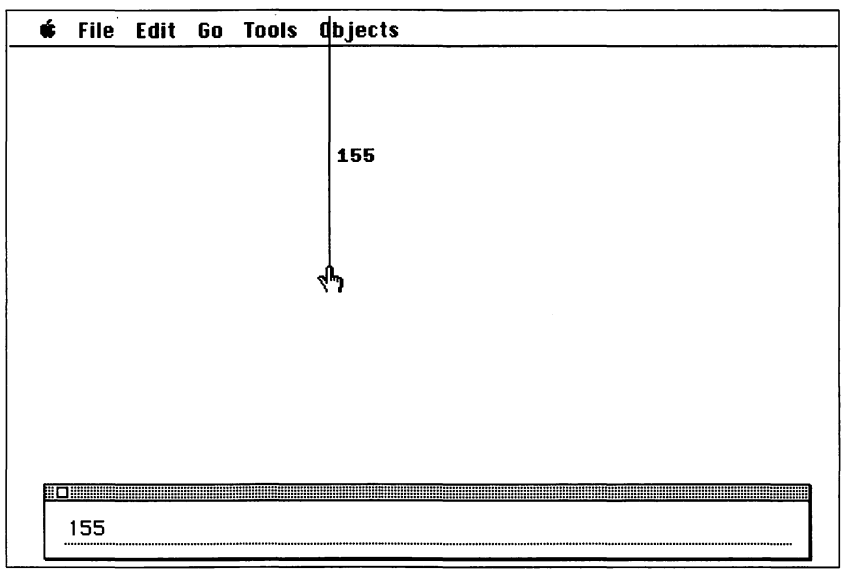


Figure 7-5. `the mouseV`.

The *the mouseLoc* Function

We now have means to find the horizontal and vertical coordinates of the mouse. What's left? The function `the mouseLoc` (see Figure 7-6) returns both coordinates as a

comma-separated pair, with the horizontal position first, followed by a comma and the vertical position.

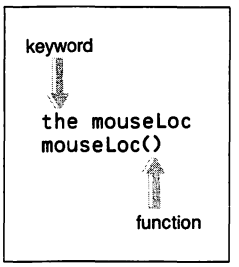


Figure 7-6. The `the mouseLoc` function syntax.

For example, if the mouse has a horizontal coordinate of 200 and a vertical coordinate of 120, `the mouseLoc` returns `200,120`. Again, you can check this easily using the Message box:

Example 3:

1. Go to your HyperTalk Bible stack.
2. Bring up the Message box and type `the mouseLoc`.
3. Press the Return key.

Result:

The coordinates of the mouse expressed in the form x,y appear in the Message box.

4. Move the mouse into the upper-left corner of the card and reenter `the mouseLoc` in the Message box.

Result:

If you get the exact corner, the Message box reports a location of `0,0`.

5. Now move the mouse (as marked by the browse tool) to the bottom right of the card and enter `the mouseLoc` in the Message box.

Result:

The Message box reads `511,341` if you locate the browse tool correctly.

Figure 7-7 shows the coordinates returned by `the mouseLoc` function. With the small-screen Macintoshes, you may get a horizontal value of 512 and a vertical value of 342. However, these points are outside the card, as is shown by the fact that the browse tool turns to an arrow when you reach these values.

Properties versus Functions

Earlier in the book, we introduced `the location` property, and here we have the function `the mouseLoc`. Both return the location of something, so you may wonder what the differences are between a property and a function. One practical difference is that most properties also are "settable." You can say `set the location of the message box to 20,20`. Functions, however, just return a value. If you try something like `set the`

`mouseLoc to 20,20`, HyperTalk rejects your attempt. A more abstract difference is that properties are aspects of particular HyperCard objects. The mouse isn't a HyperCard object, so it isn't described by properties.

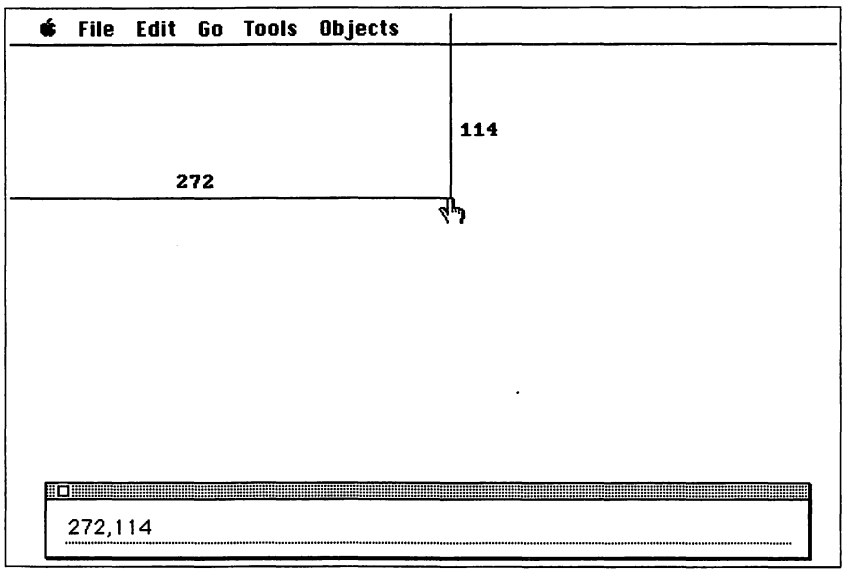


Figure 7-7. `the mouseLoc` function.

The Hot Spot

The mouse coordinates define just a single point on the screen, but the mouse cursor is quite a bit larger than that. So what, exactly, do the coordinates locate? Each cursor design has one particular point—*the hot spot*—that represents the actual mouse position. For example, the browse tool's hot spot is at the tip of the extended finger, and the fill tool's hot spot is at the tip of the pouring paint can. Figure 7-8 shows a few hot spots from Macintosh land.

The *the clickLoc* Function

The function `the clickLoc` returns the location of the mouse hot spot at the time the mouse button was last clicked. See Figure 7-9 for the syntax of `the clickLoc` function In other words, HyperTalk remembers where the last mouse click occurred, even if the mouse is moved subsequently.

If your script uses a mouse click to, say, identify a particular picture on the screen, you should use `the clickLock` rather than `the mouseLoc`. That way, even if the mouse position changes after the click, the script uses the clicked position. What if the mouse moves between pressing and releasing the mouse button? The function

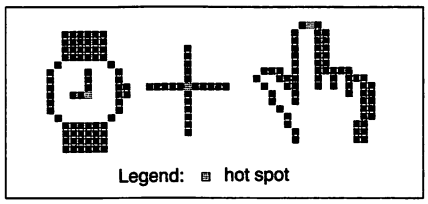


Figure 7-8. Hot spots.

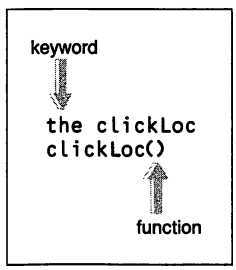


Figure 7-9. The `the clickLock` function syntax.

`the clickLoc` reports the mouse position corresponding to the moment `the mouseDown` message is sent.

Again, you can use the Message box to see how the command works.

Example 4:

1. Go to your HyperTalk Bible stack.
2. Position your mouse to your satisfaction and click it.
3. Bring up the Message box and type `the clickLoc`.
4. Press the Return key.

Result:

The coordinates of the mouse click expressed in the form x,y appear in the Message box.

5. Reposition the mouse, but don't click.
6. Type `the clickLoc` and press the Return key.

Result:

The original click coordinates are displayed again.

Typically, these mouse location functions are used to control the flow of a script. For example, a script might say, "If the horizontal mouse position is less than 100, go to card number 3" or "keep beeping until the mouse is clicked in a certain area." (Chapters 9, "Ifs," and 10, "Repeats," describe how to word those ideas in HyperTalk.) The functions also can be used to position objects. Here's an example you can run using the Message box:

Example 5:

1. Go to your HyperTalk Bible stack and open the Message box.

2. Move the mouse to the middle of the screen and click it.
3. Enter the following text in the Message box:

```
show tool window at the clickLoc
```

Result:

The tool window appears with its upper-left corner at the location you clicked.

4. Relocate the mouse, click it, and press the Return key.

Result:

The window is relocated at the new click position.

Version 1.2

The *the clickH* and *the clickV* Functions

Version 1.2 of HyperCard added the functions `the clickH` and `the clickV` to the mouse repertoire. The syntax of these functions is shown in Figures 7-10 and 7-11. Like `the clickLoc`, these functions report the location of the last mouse click. But `the clickH` reports just the horizontal location, and `the clickV` reports just the vertical location.

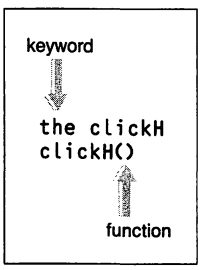


Figure 7-10. The `the clickH` function syntax.

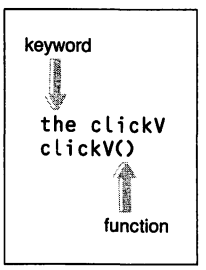


Figure 7-11. The `the clickV` function syntax.

Without these functions, you would use item 1 of `the clickLoc` to get the horizontal position and item 2 of `the clickLoc` to get the vertical position.

Here's a sample script that uses these functions to move a button around:

Example 6:

1. Go to your HyperTalk Bible stack and create a card named `Mouse`.
2. Create a button called `Move Me` on this card and give it the following script:

```
on mouseUp -- Move the button
  set the loc of me to (the clickH + 50) mod 500,¬
  (the clickV + 30 ) mod 300 + 15
end mouseUp
```

3. Click the button a few times.

Result:

With each click, the button moves diagonally down and to the right.

We used the `mod` operator to keep the button onscreen. Whenever the horizontal position exceeds 500, it gets reset to a smaller value. For instance, if `the clickH` + 50 is 535, the new horizontal location becomes `535 mod 500`, or 35. We add 15 to the vertical position to prevent the button from being hidden behind the Menu bar, which is 10 units high.

What's the Mouse Up To?

The last six functions describe the position of the mouse; the next two functions (`the mouse` and `the mouseClick`) concern the mouse's *state*. The mouse's state describes the mouse button rather than the mouse location. The button is either up or down, and those are the two possible states. HyperTalk uses the constants `up` and `down` to represent those states.

The *the mouse* Function

The function `the mouse` returns the value `up` if the mouse button is up at the moment the function is executed. And, as you've probably guessed, it returns `down` if the button is down. The syntax of `the mouse` function is shown in Figure 7-12.

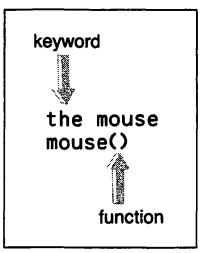


Figure 7-12. The `the mouse` function syntax.

Check it out with the Message box:

Example 7:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Type `the mouse` and press the Return key.

Result:

The word `up` appears in the Message box.

3. Type `the mouse`, hold down the mouse button, and press the Return key.

Result:

Nothing happens until you release the mouse button; then the word `up` appears in the Message box.

Up? Oops, it looks as if the mouse is having problems. Actually, the problem is with how HyperCard handles input. It doesn't accept keyboard input while the mouse button is down. As a result, the Message box command is held back until you release the mouse button, so our function `the mouse` doesn't get to check the mouse status until the button is let up again. We can get around this problem by using a button script to check the mouse status:

Example 8:

1. Go to the Mouse card of your HyperTalk Bible stack and create a new button called `Mouse Status`.
2. Enter the following handlers in the button script:

```
on mouseDown -- Mouse Status button
  put the mouse into the message box
end mouseDown

on mouseUp
  put the mouse after the message box
end mouseUp
```

3. Click the Mouse Status button once slowly. That is, press the button down, and don't release it for a while.

Result:

The Message box displays the sequence `downup`.

4. Clear the Message box and click the Mouse Status button once quickly.

Result:

If your finger is quick enough, the Message box displays `upup`. (This is fairly easy to do with Version 1.1 on a Macintosh Plus. But the added speed of Version 1.2 makes this task much more challenging.)

Here's what's happening. The on mouseDown handler tests the mouse state just after you send `the mouseDown` message by pressing the mouse button. If you are slow in removing your finger, the mouse button is still down when the mouse makes its test, and `down` is placed into the Message box. Next, the on mouseUp handler deals with the `mouseUp` message produced when you release the button. At this point, the mouse is up, so the mouse returns `up`. And what about the fast finger result? If you have a quick finger (and a responsive button), the mouse button will be back up by the time the test is made, and `up` gets typed as the first word in the Message box. Releasing the button produces the second `up`, as in the first case.

If your processor is too fast for your finger, try beginning the mouseDown handler with the command `wait 1 seconds`. This makes the Macintosh wait a second before doing the mouse test. See chapter 10, "Strings," for more about `wait`.

The function `the mouse` always returns the state of the mouse button at the time the function is called. It is possible for the mouse button to be up while a mouseDown handler is executing, even if the mouse is moved outside the object that received the `mouseDown` message. See Figure 7-13.

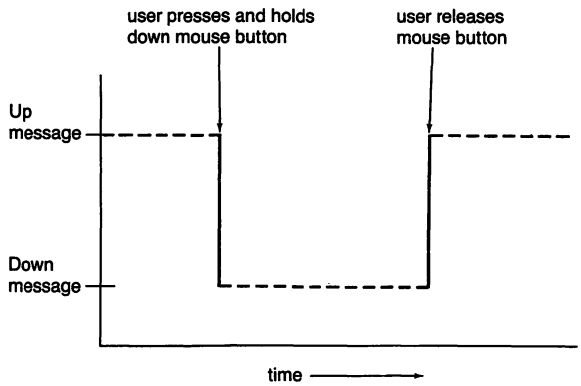


Figure 7-13. The mouse state.

The *the mouseClick* Function

The function `the mouseClick`, the syntax of which is shown in Figure 7-14, on the other hand, does have a memory. It answers the question, "Has the mouse been clicked?"

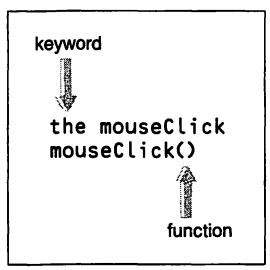


Figure 7-14. The `the mouseClick` function syntax.

This is an example of a *Boolean* function, one that is either true or false. In HyperTalk, Boolean functions return a value of true or false, where `true` and `false` are HyperTalk constants meaning exactly what they appear to mean.

A very important aspect of `the mouseClick` is what it uses as the starting point for noticing clicks. In short, `the mouseClick` looks for clicks after the handler containing the function starts or after the last use of `the mouseClick` in the handler.

The function `the mouseClick` returns true if the mouse button has been pressed since the current handler started or since the last time `the mouseclick` was evaluated, whichever happened most recently. See Figure 7-15.

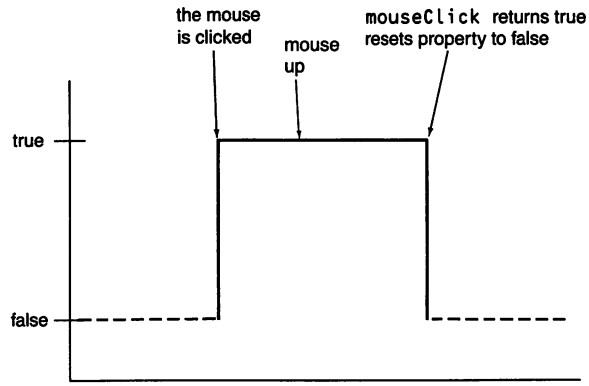


Figure 7-15. `the mouseClick`.

`the mouseClick` is another function best displayed through a script, because it is not designed to be used in the Message box.

Example 9:

1. Go to the Mouse card in your HyperTalk Bible stack and create a button called `Mouse Click`.
2. Enter the following handler in the button script:

```
on mouseEnter -- Mouse Click button
  wait 2 seconds
  put the mouseclick into the message box
end mouseEnter
```

3. Move the mouse into the Mouse Click button and wait two seconds.

Result:

The word `false` appears in the Message box.

4. Move the mouse out of the button and back into it and click the mouse before two seconds elapse.

Result:

The word `true` appears in the Message box.

Because we use `the mouseClick` in a mouseEnter handler, only a click made after the handler starts produces a `true` return value.

Typically, the mouse state functions would be used in the HyperTalk equivalent

of statements such as "if the mouse is down, show field 1" and "repeat playing this tune until the mouse is clicked."

We see more uses for all these mouse functions in the coming chapters.

The Simulated Mouse Click

The final topic for this chapter is a command rather than a function. It doesn't really come under the topic of control statement tests, but it does concern the mouse. The command is `click at`, and you can use it in a script to simulate clicking the mouse at a specified location. Figure 7-16 shows its syntax:

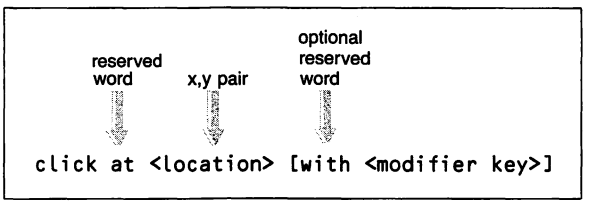


Figure 7-16. The `click at` command syntax.

The ⟨location⟩ is the usual HyperTalk location: the horizontal (x) and vertical (y) coordinates separated by a comma; if you knew a button was located at 30,40, you could type the command `click at 30,40` and get the same effect as clicking the button. You can use any property, container, or function that provides a location. Sometimes you use one of the modification keys (the Shift key, the option key, or the command key) when using the mouse. The optional part of the command lets you specify the modification key. For example, you can use the Option-key-click combination to clone a button. To simulate this in a script, you could say `click at 30,40 with OptionKey`.

Here's a simple example in which clicking one button serves to click a second button:

Simulating Double-Clicking Buttons

HyperCard's buttons are designed to react to a single click of the mouse button. It is possible to create a handler that detects a double click. The following handler waits 30 ticks for the second click and times out if that second click doesn't occur.

```
on mouseUp
   put the ticks into startTicks
   repeat until the mouseClick
       if the ticks - startTicks > 30 then exit mouseUp
   end repeat
   -- Your instructions for the double-click go here
end mouseUp
```

Example 10:

1. Go to the Mouse card of your HyperTalk Bible stack and create a button called `Clyde`.
2. Enter this handler in Clyde's script:

```
on mouseUp -- Clyde button
  beep 3
end mouseUp
```

3. Next, create a new button called `Click Clyde`.
4. Enter this handler in Click Clyde's script:

```
on mouseUp -- Clyde button
  click at the location of button "Clyde"
end mouseUp
```

5. Click the mouse in Click Clyde.

Result:

The Macintosh beeps as Clyde demands.

So you click Click Clyde, and Click Clyde clicks Clyde. What happens after Click Clyde clicks Clyde? Control goes back to the handler in Click Clyde. You can see this (actually, you hear it) by slightly modifying the last example:

Example 11:

1. Go to the Mouse card of your HyperTalk Bible stack and open the script for Click Clyde.
2. Add a line to the handler so that it reads as follows:

```
on mouseUp
  click at the location of button "Clyde"
  play "boing"
end mouseUp
```

3. Click the Click Clyde button.

Result:

You hear the beeps from the Clyde script and then the boing from the Click Clyde script.

So after clicking at the other button and letting it do its bit, the Click Clyde handler finishes its own business.

Let's take advantage of HyperCard's object heirarchy and show a more powerful manifestation of simulation power.

Example 12:

1. Go to the Mouse card of your HyperTalk Bible stack.
2. Open the Mouse card script and add this text:

```
on mouseWithin
  click at the mouseLoc
end mouseWithin
```

3. Move the mouse over various buttons on the card.

Result:

Moving a mouse into a button activates the mouseUp handler in it.

As long as the mouse is within a button, the `mouseWithin` message is sent to the button. However, our buttons don't have a handler for this message, so it is passed to the card level. There, it animates our on mouseWithin handler, which then clicks at the mouse location. Because the mouse is in a button, this amounts to clicking the button repeatedly until the mouse leaves the button.

Try moving the mouse into the Mouse Status button. Because the simulated click is faster than a mechanical click, you get an `upup` report in the Message box.

Summary

HyperTalk offers tight control over the mouse interface. The various mouse messages let you set up handlers for a variety of situations. The functions `the mouseH`, `the mouseV`, and `the mouseLoc` report the mouse location, and `the clickLoc` reports the location of the last mouse click. The functions `the mouse` and `the mouseClick` report on the mouse state: is it up or down, and has it been clicked recently? Finally, the HyperTalk command `click at` lets you simulate clicking a mouse from within a script.

Review Questions

1. What are two ways of finding the horizontal mouse coordinate?
2. What's the difference between `the mouseLoc` and `the clickLoc`?
3. Can `the mouse` be used rather than `on mouseDown`?
4. Match the function with its return value:

a. `the mouseH`	f. mouse coordinates
b. `the mouse`	g. coordinates of last click
c. `the mouseClick`	h. horizontal mouse coordinate
d. `the mouseLoc`	i. mouse button status (up or down)
e. `the clickLoc`	j. click status (true or false)

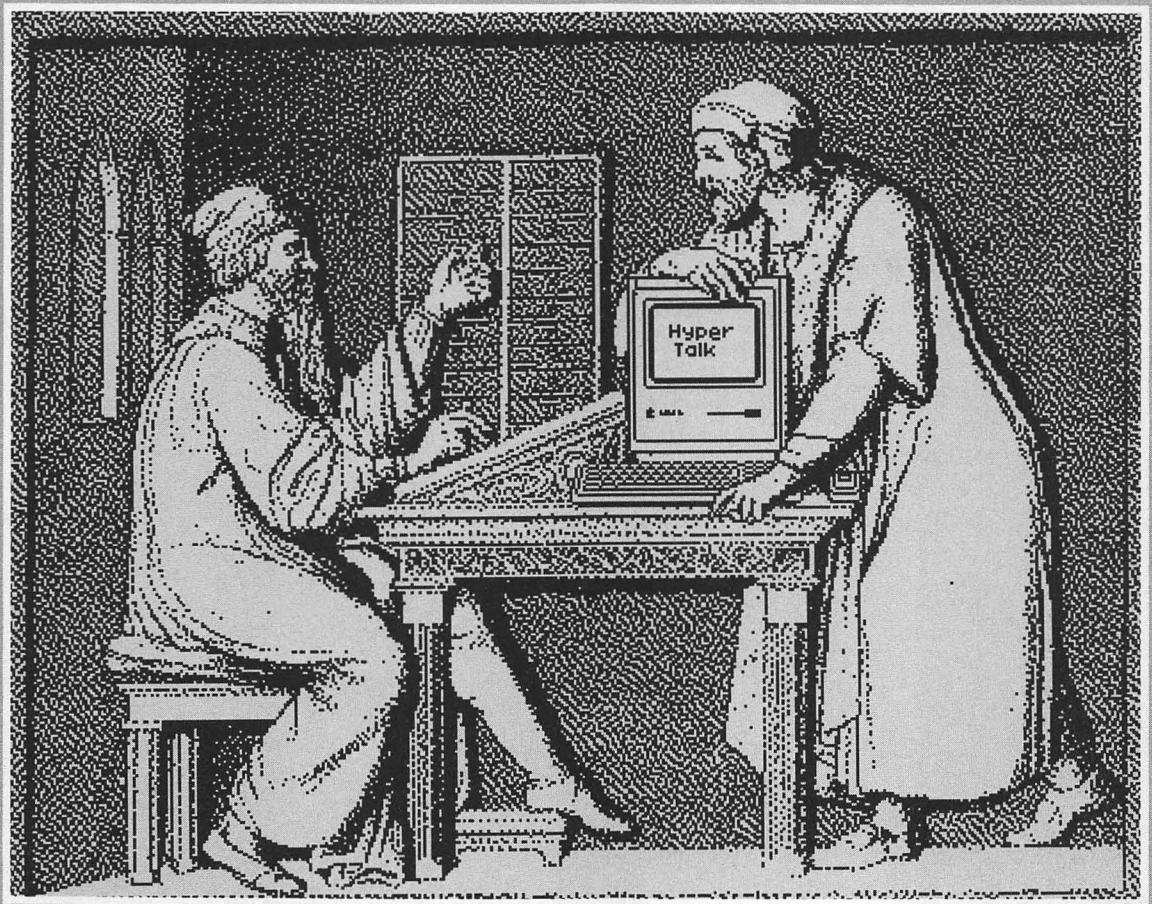
5. Can `the mouseClick` be used more than once in a handler?
6. How can you simulate clicking the mouse at the location of the last mouse click, but this time, with the option key down?
7. Example 12 causes a button to be continually activated as long as the mouse is within the button. How would you change the example so that the button is activated just once?

Exercises

1. Create a button script that causes the location of the last mouse click and the current mouse position to be displayed in the Message box. Try pressing the button down when the mouse is at one end of the box and releasing the button when the mouse is at the other end. The results demonstrate that `the clickLoc` remembers the mouse location at the time the button is pressed, not when it is released.
2. Assume the card script of Example 12 is removed. Use the `click at` command to create a button script that reports the mouse position to the Message box as long as the mouse is in the button. Can this same aim be accomplished without using the `click at` command?

Chapter 8

Keyboard



Quick View

Commands
ask
ask password
answer
type

Keywords
left
right

Functions
the shiftKey
the optionKey
the commandKey

Messages
returnKey
enterKey
tabkey
arrowkey
controlKey
functionKey

Concepts
Key states
Keyboard input

Chapter 8

Keyboard

The mouse is a great way to communicate with a HyperCard stack, but it can't carry the whole load. Sometimes you need to turn to the keyboard. Perhaps you need to detect whether the option key is down, you want the user to type an answer to a question, or you want to process one of the arrow keys. HyperTalk has several keyboard functions, messages, and commands to help you. To experience the full benefit of many of these features, you need to use the `if` statements and `repeat` loops of the next two chapters. In this chapter, we concentrate on the basics.

HyperTalk handles keyboard input quite differently from languages like BASIC, Pascal, or C. Those languages provide facilities for reading individual keystrokes. HyperTalk takes a higher-level approach. The `ask` command, for example, provides a dialog box for obtaining user input, or the user can be directed to enter data in fields. HyperTalk tends to work with chunks of input rather than with individual keystrokes, but there are some keys, such as the option key or the right arrow key that have special meaning to HyperTalk, and those keystrokes you can check for.

Keyboard Functions

Three keys—the Shift key, the option key, and the command key—play a special role because they modify the meaning of other keys. They affect, for example, how HyperCard interprets the mouse button. For instance, when you are using the button tool, you can press the mouse button down and drag a card button around, but by pressing the option key simultaneously, you can duplicate a button. These keys also are used a lot for the paint modes.

Hypertalk provides three functions to describe the state of these three keys. (See Figure 8-1 for the keyboard functions syntax.) The functions are `the optionKey`, `the shiftKey`, and `the commandKey`. Each function returns the HyperTalk constant `up` if the corresponding key is in the up position when the function is executed. If the key is down, the corresponding function returns `down`.

In short, both functions work like `the mouse`. Like `the mouse`, too, they usually are used with the control statements covered in chapters 9, "Ifs," and 10, "Repeats."

```
the shiftKey
shiftKey()

the optionKey
optionKey()

the commandKey
commandKey()
```

Figure 8-1. The keyboard functions syntax.

Here, we just take a quick visit to a HyperCard stack to verify how these functions work.

Testing the shift and option keys is simple. Just follow these steps:

Example 1:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Type `the shiftKey` into the Message box and press the Return key.

Result:

The HyperTalk constant `up` appears in the window.

3. Repeat step 2, but hold down the Shift key while pressing the Return key.

Result:

This time `down` appears in the window.

4. Repeat steps 2 and 3, but use the option key and the function `the optionKey`.

Result:

The Message box shows `up` if the option key is up and `down` if the key is down.

You can try the same approach with the function `the commandKey`, but this time you'll have a problem: The Return key doesn't function normally in HyperTalk when the command key is down. However, you can turn to a script.

Example 2:

1. Go to your HyperTalk Bible stack and use Command-N to create a new card called `Keyboard`.
2. Create a button called `Command Status` and enter the following text:

```
on mouseWithin -- note we aren't using the usual mouseUp message
  put the commandKey into the message box
end mouseWithin
```

3. Move the mouse into the Command Status button and press and release the command key a few times.

Result:

While the key is down, the word `down` appears in the Message box, and while the key is up, `up` appears in the Message box.

Recall that the `mouseWithin` message is sent repeatedly as long as the mouse is in the button. This script, therefore, is run repeatedly as long as you keep your mouse in the Command Status button. Each time the script runs, it reports the state of the command key.

Even though we don't cover `if` statements until the next chapter, we can't resist showing you how they work hand in hand with these keyboard functions.

Example 3:

1. Go to the Keyboard card of your HyperTalk Bible stack and create a button named `Command Boing`.
2. Put the following script in the button:

```
on mouseWithin -- Command Boing button
  if the shiftKey is down then play "boing" c
  if the optionKey is down then play "boing" d
  if the commandKey is down then play "boing" e
end mouseWithin
```

3. Place the mouse into the Command Boing button and press and release the Shift, option, and command keys.

Result:

Different boing notes sound as each key is pressed.

The `if` statements work just as they read. If the Shift key is down when a `mouseWithin` message is sent, the `play "boing" c` command is executed. If you hold a key down too long, you get several boings because several `mouseWithin` messages are sent before the key is released.

The User Interface Functions

Sometimes your stack needs to get information from the user: a number, perhaps, or a word. The usual Macintosh method is to put up a dialog box in which the user can type a response or use the mouse to select a response. Building a dialog box from scratch requires a lot of programming. Fortunately, HyperTalk supplies built-in commands to provide these services. The commands are `ask`, `ask password`, and `answer`. They make it simple to obtain information from a user. We introduced `ask` and `answer` in chapter 3, "Audio and Visual Effects," but in the interests of gathering the keyboard commands in one chapter, we look at all three of these dialog boxes now.

The *ask* Command

The `ask` command (for syntax, see Figure 8-2) displays your question in a dialog box (which we call an *ask box*) in the middle of the screen, and it provides a space into which users can type answers. Users then have a choice between clicking an OK button or a Cancel button. Pressing the Return or Enter key has the same effect as clicking OK. That effect is to place the answer into the global variable `it`. Clicking the Cancel button puts the empty string (no characters and represented by the HyperTalk constant `empty`) into `it`.

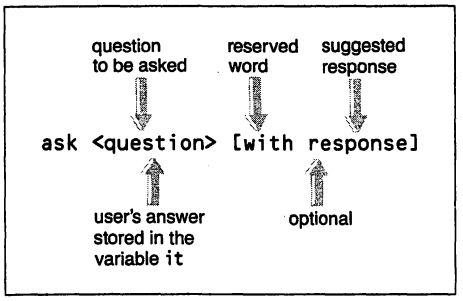


Figure 8-2. The `ask` command syntax.

The `ask` command uses the Chicago 12-point font to display the question and the user's response. This font size and the ask box design require that you limit the question to about forty characters or less. (Because the Chicago font uses proportional spacing, the exact figure depends on the characters you use.) The `question>` may be a string or a container or function providing a string. We return to the optional `with <response>` part of the syntax shortly, but first let's test the simplest use.

Example 4:

1. Go to the Keyboard card in your HyperTalk Bible stack and create a button called `I Ask`.
2. Put the following script into the button:

```
on mouseUp -- I Ask button
  ask "What's your favorite currency?"
  put it into the message box
end mouseUp
```

3. Click the I Ask button.

Result:

The ask box shown in Figure 8-3 appears on the screen.

4. Type a response and click the OK box.

Result:

Your response reappears in the Message box.

As Figure 8-3 suggests, HyperTalk places the ask box into the middle of the card on a small-screen Macintosh.

Clicking the OK button causes `ask` to accept your typed answer and to assign it to the global variable `it`. Then, the `put` command places your answer into the Message box. Notice that the OK button has a double outline. That means that pressing the Return or Enter key also selects that button.

What happens if you choose the Cancel button? Does that cancel the command or the whole script? Let's find out.

Example 5:

1. Return to the Keyboard card of your HyperTalk Bible stack and open the I Ask script.

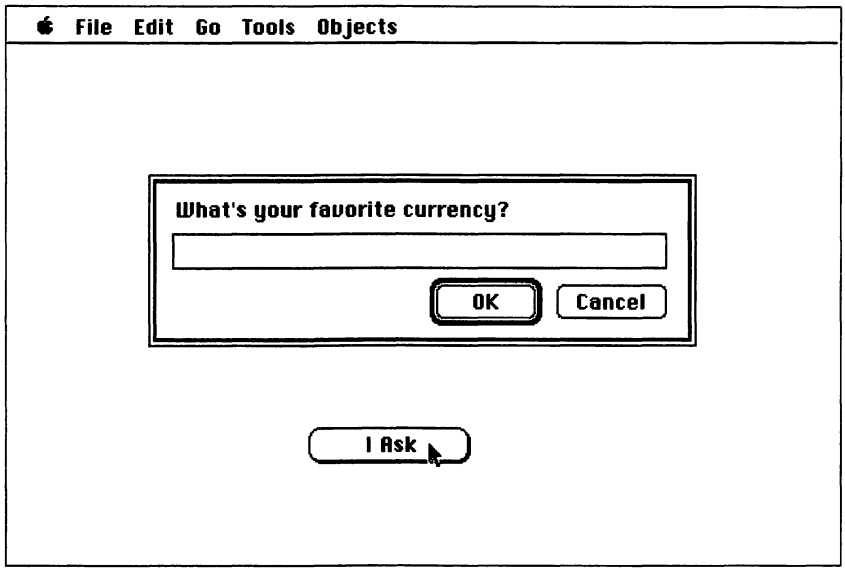


Figure 8-3. The `ask` Message box.

2. Modify it to read as follows:

```
on mouseUp
  ask "What's your favorite currency?"
  put it into the message box
  dial "3 2"
end mouseUp
```

3. Click the I Ask button, type an answer, and click the Cancel button.

Result:

The Message box, if not empty, is cleared, and the Macintosh makes the "3 2" dial tones.

The script following the `ask` command is not canceled, but the variable `it` is assigned the HyperTalk constant `empty`. What if you do want the rest of the script canceled? Use the `if` statement to see whether `it` was set to `empty`: `if it is empty then blah blah`.

Next, let's check out the optional `with <response>` part of the `ask` command. This part lets you suggest a response—in the form of a string or a container or function providing a string—to the question by placing the suggested response into highlighted form in the ask box's text response box. Then, users can click the OK button to accept your suggested response or type another answer.

Example 6:

1. Return to the Keyboard card in your HyperTalk Bible stack and reopen the I Ask script.
2. Modify it to read as follows:

```
on mouseUp
  ask "What's your favorite currency?" with "Pengos"
  put it into the message box
end mouseUp
```

3. Click the I Ask button.

Result:

The ask box appears as shown in Figure 8-4.

4. Click the OK box.

Result:

`Pengos` appears in the Message box, as shown in Figure 8-4.

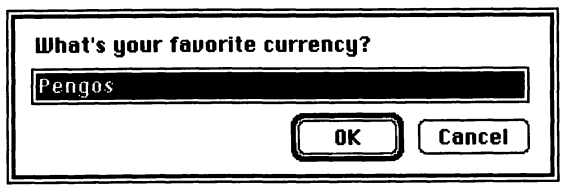


Figure 8-4. The `ask` dialog box with suggested default answer.

As promised, clicking the OK box chooses the preselected response. (The infamous Hungarian pengo showed what inflation really is about when, in 1946, it took 130 quintillion paper pengos to equal one gold pengo.)

The *ask password* Command

The `ask password` command has the same syntax as the `ask` command (see Figure 8-5), and it works much the same.

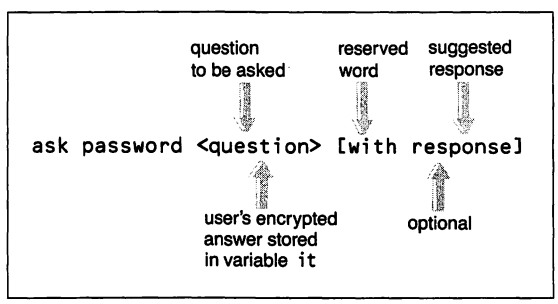


Figure 8-5. The `ask password` command syntax.

The difference is that the `ask` command assigns your answer to the variable `it`, but the `ask password` command encrypts your answer into a number and assigns that number to `it`. Encrypting is the translation of ordinary text to some other form by

using some form of key. For example, by using the simple code of replacing each letter by the one alphabetically preceding it, you can produce HAL as the encrypted form of IBM. HyperCard uses a more sophisticated key that translates text to a 10-digit integer.

We can check the encryption easily enough.

Example 7:

1. Go to your HyperTalk Bible stack and bring up the Message box.
2. Enter the following into the Message box:

```
ask password "Who goes there!"
```

Result:

The ask box shown in Figure 8-6 appears on screen.

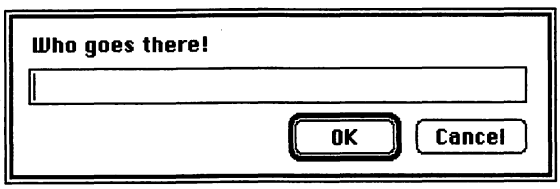


Figure 8-6. The `ask password` dialog box.

3. Type `greyfog` into the ask box and click OK.
4. Enter `it` in the Message box.

Result:

The ask box disappears after you click OK, and when you enter `it`, the number `1939083412` appears in the Message box.

Here, 1939083412 is the encrypted form of greyfog.

So what good does this do you? No good, until you learn how to use the `if` command to compare this response to a stored response. Luckily, we already peeked at `if`, so we can now show a simplified use of `ask password`. We use it to block access to a particular card unless users know the password. We find the encrypted form of a password and store it in a field. Then we use `ask password` to make users enter passwords. Next, we compare users' encrypted responses to the stored one. If they don't match, we dump the users out of the card. Even if the users snoop to see what the encrypted password is (1939083412, for instance), that doesn't tell them what password produces the encrypted value. (If they type 1939083412, for example, that gets encrypted to some other value.) The main point here is that the encryption method makes it easy to go from text to the encrypted value but virtually impossible to go from the encrypted value back to the original.

Example 8:

1. Go to your HyperTalk Bible stack and use Command-N to create a new card called `Restricted`.
2. Create a card field and name it `Password`.

3. Enter the following into the Message box:

```
ask password "Who?"
```

4. Type `greyfog` into the ask box and click OK.

```
greyfog
```

5. Enter the following into the Message box:

```
put it into card field "Password"
```

6. Now enter

```
hide card field Password
```

7. Open the card script and enter

```
on openCard
  hide card field "Password"
  ask password "Your code name, intruder!"
  if it <> first word of card field "Password" then go to first card
end openCard
```

8. Go back one card, and then return to this card.

Result:

The ask box shown in Figure 8-7 appears on screen.

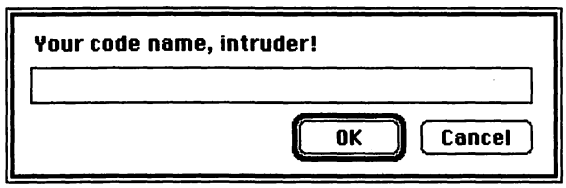


Figure 8-7. Using `ask password`.

9. Respond with a name other than greyfog.

Result:

You are sent to the first card of the stack.

We can paraphrase the `if` command like this: if the user's encrypted response is not equal to the response stored in card field Password, go to the first card. This example shows how `ask password` can be used to restrict access to particular cards or stacks. (Actually, it's pretty simple to evade this particular guardian. For example, you can type Command-period to cancel the `ask password` command. Remember, all we claimed we would do is supply the rudiments.)

One interesting point is that Greyfog and GREYFOG work the same as greyfog; the encryption process ignores uppercase and lowercase.

Another use for `ask password` is to establish a set of quiz questions with a field holding the encrypted answers.

Like `ask`, `ask password` optionally accepts a suggested answer in the ask box. You wouldn't want to suggest the correct answer, of course, but using this feature might lull users into thinking they're dealing with an ordinary ask dialog box.

Ask Password Delights

Your password can be anything that fits in the dialog box. That means that there are many more possible passwords than there are available codes. The number of eight-letter combinations (like ASDFGHJK) alone exceeds the number of codes. However, HyperTalk doesn't simply ignore extra characters: AAAAAAA, AAAAAAAA, AAAAAAAAA, and AAAAAAAAAA all have distinct representations. This, in turn, implies that two or more words can have the same encrypted value. The chances of stumbling across such a pair by accident is quite remote, however.

The *answer* Command

If the `ask` command asks your questions, does the `answer` command answer them? Not quite. Like `ask`, the `answer` command lets you pose a question, but instead of giving the user a free rein to respond, `answer` presents the user with a dialog box containing a set of buttons listing acceptable responses. (We call this form of dialog box an answer box.) The user then clicks the button of his choice. Figure 8-8 shows the syntax for the `answer` command.

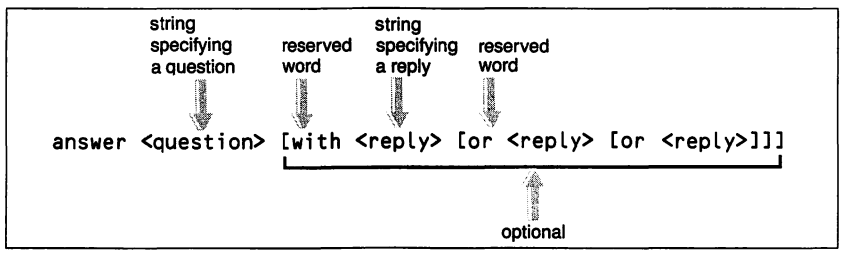


Figure 8-8. The `answer` command syntax.

Note that the syntax provides for from zero to three suggested answers. Also note that the ⟨question⟩ and ⟨reply⟩ options are entered in the form of strings or containers or functions providing a string. First, let's see what happens when we omit the suggested answers.

Example 9:

1. Go to the Keyboard card of your HyperTalk Bible stack and create a new button called `Answer Me`.
2. Place the following script into the button:

```
on mouseUp -- Answer Me button
  answer "Want some broccoli?"
  put it into the message box
end mouseUp
```

3. Click the Answer Me button.

Result:

The answer box shown in Figure 8-9 appears on screen.

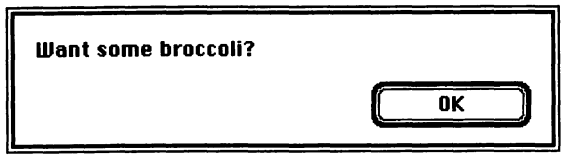


Figure 8-9. An answer dialog box (default answer button).

4. Click the OK button.

Result:

OK appears in the Message box.

If you fail to use the `with <response>` option to suggest answers, the `answer` command provides an OK button. It has a double outline, which means you can select it by pressing the Return or Enter keys or by clicking the mouse in the button. The selection is assigned to the global variable `it`, and the script then copies it in the Message box.

One common use for this form of answer is to bring some information to the attention of the user: `answer "This feature not yet implemented"`. After reading the message, the user clicks OK and goes on. This is an example of an alert box.

Next, let's at least give the prospective broccoli eater some options.

Example 10:

1. Return to the Answer Me button.
2. Modify its script to read as follows:

```
on mouseUp -- Answer Me button
  answer "Want some broccoli?" with "Yuk!" or "Yum!" or "Maybe"
  put it into the message box
end mouseUp
```

3. Click the Answer Me button.

Result:

The answer box shown in Figure 8-10 appears. Note that the Maybe button, the final choice, has a double outline.

4. Click the Yum! button.

Result:

`Yum!` appears in the Message box.

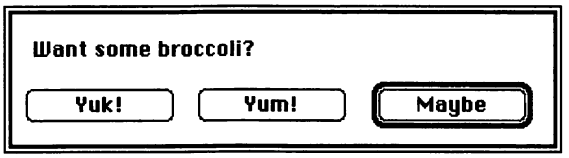


Figure 8-10. An `answer` dialog box with three answers.

In general, whichever answer you click is assigned to the global variable `it`, and the final suggested choice gets the double outline, indicating that it can also be selected by pressing the Return or Enter key. In other words, the final choice is the default response.

These examples show how the `answer` command works but not how to use it. Typically, you would use an `if` construction that selected a course of action based on the user's response. For example, you might use something along these lines: `"answer proceed?" with Cancel or OK`. This makes OK the default response, because it is last in the list. Then, you have the script check to see whether the global variable `it` is set to Cancel and to quit if it is. So the `answer` command is another HyperTalk facet intimately associated with the HyperTalk control statements.

Keyboard Messages

HyperTalk recognizes six system messages generated by keystrokes: `returnKey`, `enterKey`, `tabKey`, `arrowKey`, `controlKey`, and `functionKey`. (Many Macintosh keyboards do not have all these keys. If you don't have an SE or a Macintosh II, you probably don't have a Control key, and if you don't have a Macintosh II, you probably lack a set of function keys.) Each message is generated when the corresponding key is pressed. Unlike `mouseUp` and many of the other messages we've discussed, these messages bypass the button and field levels of the HyperCard message hierarchy (see chapter 2, "Objects"). Instead, they enter the system at the card level. See Figure 8-11.

Keep in mind that that although the names are quite similar to keyboard function names like `shiftKey`, keyboard messages are used differently from keyboard functions. The keyboard functions return values (`up`, `down`, `true`, `false`) and are used inside message handlers. The keyboard messages, however, are used to initiate handlers. The following shows the difference in how functions and messages are used:

```
on enterKey -- do the following when the enterKey message is sent
  put the shiftKey into the msg box -- check status of shift key
end enterKey
```

You cannot interchange the roles of `enterKey` and `shiftKey` in this example. Also note that this message handler would be placed in a card, background, or stack script because the keyboard messages are not sent to buttons and fields.

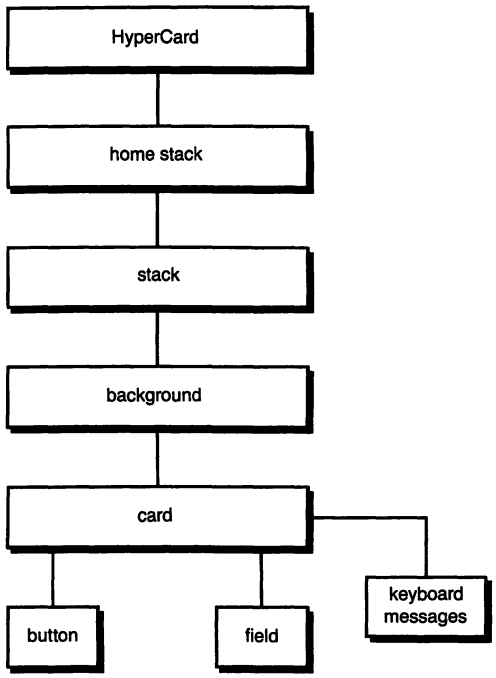


Figure 8-11. Keyboard messages and the hierarchy.

The *returnKey*, *enterKey*, and *tabKey* Messages

These three messages work the same way. First, each is generated when the corresponding key is pressed. Second, each enters the hierarchy at the card level. Third, each is ignored if you are in a text mode. This means you can redefine the effect of pressing these keys without affecting their normal HyperCard meanings when they are used to enter text in a field or dialog box or to edit a script.

Let's try these messages.

Example 11:

1. Go to your HyperTalk Bible stack and create a new card called `Keyboard Messages`.
2. Enter the following text in the card script:

```
on returnKey
  play "boing" tempo 200 f f c
end returnKey

on enterKey
  play "boing" tempo 150 g g d
end enterKey
```

continued

```
on tabKey
  play "boing" tempo 100 a a e
end tabKey
```

3. Press the Return, Enter, and Tab keys.

Result:

Your Macintosh plays a different tune for each of the three keys.

Great! These keys have become musical, but have they lost their normal abilities?

4. Open a field and enter text, using the Return key to start new lines and the Enter key to terminate the text entry.

Result:

The Return and Enter keys function normally.

When you are entering text in a field or script, the Return and Enter keys function normally, even if you are using message handlers for them.

5. Open the Message box.
6. Type the date and press the Return key.

Result:

HyperTalk plays a short tune, and the words `the date` remain in the message window.

This is not the result we expected. Because our handler traps the `returnKey` message, the Return key no longer terminates input to the Message box!

What if you want to redefine the effect of the Return key yet still have it work in the Message box? This is a job for the `pass` command (see chapter 6. "Messages"). Remember that this keyword can be used to pass a message on after a script has handled it. Let's see how that works.

Example 12:

1. Return to the Keyboard Messages card of your HyperTalk Bible stack and reopen the card script.
2. Modify the `on returnKey` script to read as follows:

```
on returnKey
  play "boing" tempo 200 f f c
  pass returnKey
end returnKey
```

3. Open the Message box and type the date.
4. Press the Return key.

Result:

HyperTalk plays a tune, and then places the current date into the Message box.

This time, the script detects the `returnKey` message, executes the `play` command, and then passes the `returnKey` message on to perform its normal function, which, in this case, is to cause the command in the Message box to be sent.

The *arrowKey* Message

At first glance, the `arrowKey` message is much like the preceding three. You can use it the same way.

Example 13:

1. Return to the Keyboard Messages card of your HyperTalk Bible stack and reopen the card script.
2. Add the following text:

```
on arrowKey
  beep
end arrowKey
```

3. Press alternately any arrow key and the Enter key.

Result:

The Enter key (unless you removed the script from Example 12) plays a tune, and any one of the arrow keys instigates a beep.

The tricky aspect of the `arrowKey` message is that you can use it to detect which arrow key was pressed. Each `arrowKey` message is accompanied by one parameter, which can be any one of the four HyperTalk constants `up`, `down`, `left`, or `right`. That messages, like functions, can have parameters is a standard HyperTalk feature. However, message parameters more typically come with user-defined messages than with system messages. The `arrowKey` message is an exception, so we have to preview some material found in chapter 14, "Advanced Commands and Functions," on user-defined messages and functions.

There are, in fact, two methods. One is to use the function `the param of`. The simpler method, and the one we use here, is to append a parameter name to the `on arrowKey` line. In essence, that name becomes a variable that is assigned the parameter passed with the message. Let's see how that works:

Example 14:

1. Go to the Keyboard Messages card of your HyperTalk Bible stack and open the card script.
2. Insert the following text:

```
on arrowKey direction -- "direction" is the parameter name
  put direction after the message box
end arrowKey
```

3. Press the various arrow keys.

Result:

When you press the left arrow key, for instance, the HyperTalk constant `left` appears in the Message box.

When you press the left arrow, the message `arrowKey left` is sent; `arrowKey` is the message, and `left` is the parameter. The on arrowKey handler "traps" this mes-

sage and assigns the value `left` to direction. Then, the script copies the value of `direction` into the Message box. We didn't have to use the word `direction` to represent the parameter. We could have used `which` or `suzie` or any other name that satisfies HyperCard's rules for variable names.

This example shows how to determine which key is pressed. In the next chapter, we show how to use that information.

The *functionKey* Message

The extended keyboard that comes with the Macintosh II has a row of keys across the top labeled F1 through F15; these are called the function keys. HyperTalk handles them in the same fashion as it handles the arrow keys. Typing a function key sends a `functionKey` message accompanied by a parameter; in this case, the parameter is an integer in the range 1 through 15, indicating which function key was pressed.

Example 15:

1. Go to the Keyboard Messages card of your HyperTalk Bible stack and open the card script.
2. Insert the following text:

```
on functionKey number
  play "harpsichord" 59 + number
end functionKey
```

3. Type the various function keys.

Result:

Each function key produces its own note.

Because the parameter, in this case, is a number, we can use it to specify which note is played. Pressing the F1 key, for instance, transmits the parameter 1, causing note 60 to be played.

The *controlKey* Message

The enhanced keyboard also has a Control key. This key, like the Shift key, is used in conjunction with other keys. For example, Control-C means "press the C key while the control key is depressed." When you press a combination such as Control-C, a `controlKey` message is sent, along with a parameter in the range 1 through 255 to indicate which key was struck.

Example 16:

1. Go to the Keyboard Messages card of your HyperTalk Bible stack and open the card script.
2. Insert the following text:

```
on controlKey number
  put number into the message box
end controlKey
```

3. Type the various control key combinations.

Result:

You see the number associated with each key.

The keys a through z are represented by the numbers 1–26. However, the associations are not unique. For example, the Control-Enter combination and the Control-C combination both use 3 as a parameter.

Keyboard Simulation: The *type* Command

In a script, you can simulate typing on the keyboard by using the `type` command. The syntax looks like that shown in Figure 12.

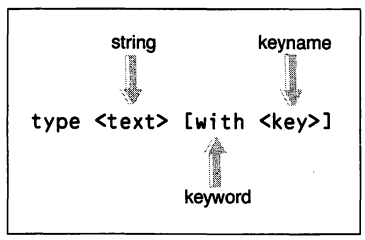


Figure 8-12. The `type` command syntax.

The command types `<text>`—which can be a string constant or a container or function returning a string—at the *current text insertion point*. The insertion point locates where the next character you type appears. Normally, it's marked with the flashing vertical bar cursor. For example, if you click in a text field, the flashing vertical bar cursor appears at the click location. When you run an `ask` command, the flashing vertical bar cursor (and insertion point) appears in the dialog box answer window. If you select text by highlighting it, you don't see a flashing vertical bar cursor, but there is still a current insertion point: the beginning of the highlighted area. If you haven't selected any text and don't have the flashing vertical bar cursor showing, the beginning of the Message box, if open, is the default current text insertion point. (If you've selected Blind Typing on the User Preferences card, the beginning of the Message box is the default current text insertion point even if the Message box is not visible.)

Because `type` puts text at the current insertion point, it places text at the same location that text would appear if you were to type it from the keyboard.

The optional `with <key>` part simulates typing with `key` depressed; your

choices for `key` are `shiftKey`, `optionKey`, and `commandKey`. So typing `m` with `commandKey` is the same as typing Command-M. Therefore, it brings up the Message box.

Let's try `type`.

Example 17:

1. If you have Blind Typing selected on your User Preference card in the Home stack, deselect it for this example.
2. Go to the Keyboard card of your HyperTalk Bible stack and create a new button called `Type`.
3. Put this script in the button:

```
on mouseUp -- Type button
  type "Help! I'm trapped inside a HyperTalk button!"
end mouseUp
```

4. Open the Message box and click the Type button.

Result:

The message `Help! I'm trapped inside a HyperTalk button!` appears in the Message box one character at a time.

One reason to use `type` rather than `put` is to produce this typing effect.

4. Close the Message box and click the Type button.

Result:

You hear a series of beeps.

If there is no current text insertion point, the Macintosh beeps once for each character in the string being typed, just as if you were attempting to type at the keyboard. (If Blind Typing is selected, the text goes in the Message box whether or not the box is showing.)

5. Create a new card field, click the mouse in the field to place the I-beam cursor there, and then click the Type button.

Result:

The text `Help! I'm trapped in a HyperCard button!` appears in the field.

This illustrates the fact that the text is typed at the current text insertion point.

Version 1.0.1 Private Access Bug

There is a potentially fatal error that prevents a stack from opening again after being quit in Version 1.0.1. The bug occurs when a new stack is assigned a password via the Protect Stack menu item, the stack is quit and relaunched, and the Private Access is set via Protect Stack. The stack is quit again and is now inaccessible to HyperCard. The workaround is never to allow users to set the private access feature after setting a password. Version 1.1 of HyperCard fixes this bug.

Summary

HyperTalk provides several tools for dealing with keyboard input. The functions `the shiftKey`, `the optionKey`, and `the commandKey` report the states of those keys: `up` or

down. This is handy information, for these keys often are used to modify the behavior of the mouse. The commands ask, ask password, and answer make it simple to create dialog boxes to solicit information from the user. And the messages returnKey, enterKey, tabKey, arrowKey, functionKey, and controlKey let you initiate actions with the corresponding keystrokes. Finally, you can use the type command to simulate typing at the keyboard.

Review Questions

1. What's wrong with the following example?

```
on optionKey
  beep
end optionKey
```

2. Which of the following are function names and which are message names?

   a. commandKey
   b. shiftKey
   c. returnKey
   d. controlKey
   e. optionKey
   f. tabKey

3. What's the difference between ask and ask password?
4. What happens if you put an on returnKey message handler into a button script?
5. What parameter is sent with each of the following keystrokes?

   a. left arrow
   b. F5
   c. down arrow

6. Why are keyboard messages used in button scripts?
7. The text of a type command appears at which location?

   a. always in the Message box
   b. at the current location of the mouse
   c. at the current text insertion point

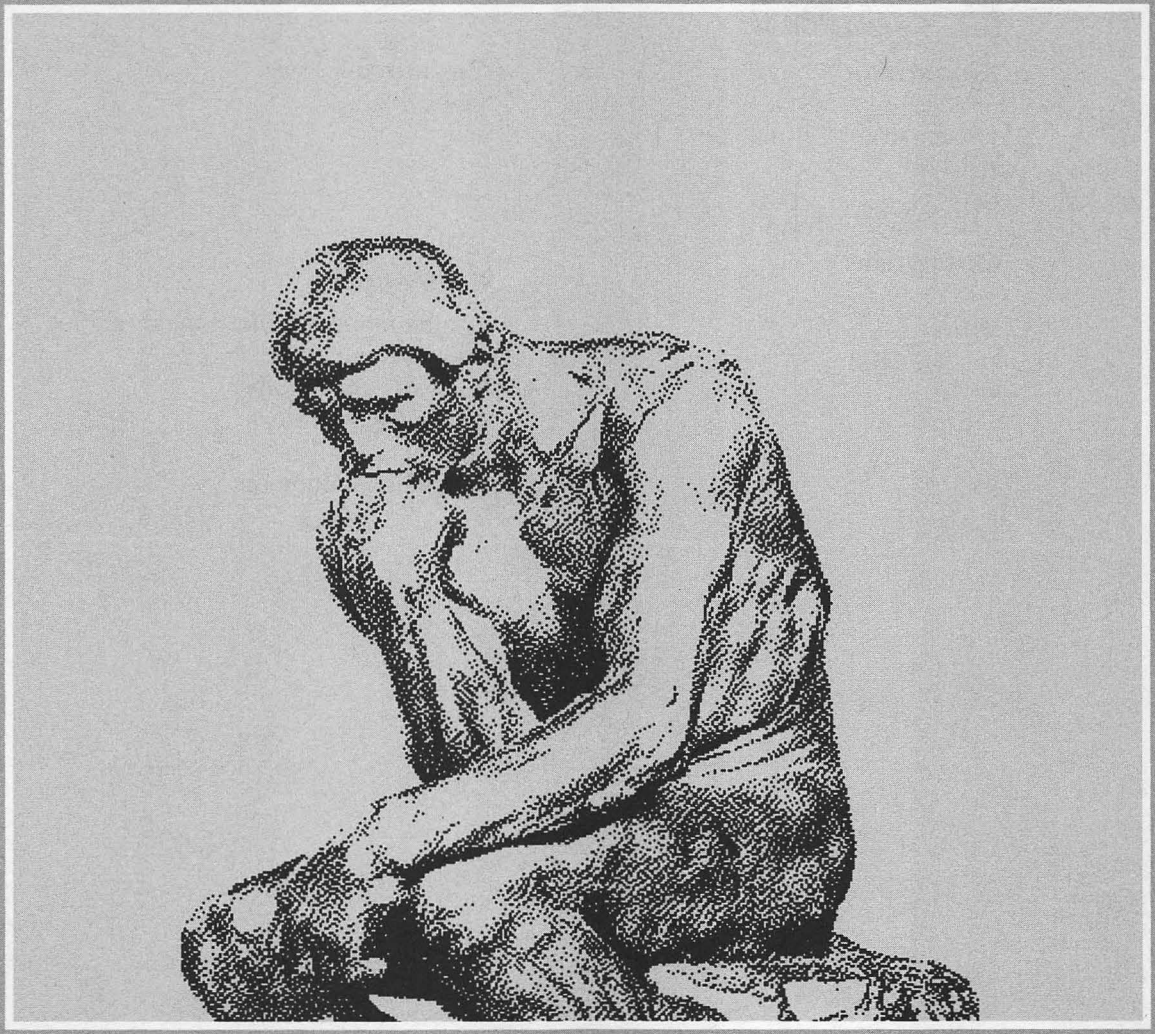
Exercises

1. Create a button that beeps if the mouse enters it while the Shift key is down and that plays the harpsichord sound if the option key is down when the mouse leaves the button. Use the if form shown in Example 3.
2. Write an answer command that asks users whether they want to continue, repeat, or quit. Make the continue option the default.

3. Create a simplified quiz button that asks for the capital of California. Have it place `Right!` into the Message box if the response is `Sacramento` and `Wrong!` otherwise. Have the button compare encrypted values instead of testing for `Sacramento` directly.
4. Write a script that causes the up arrow key to sound one note and the down arrow key to sound a different note. Also, make sure the usual uses for these keys still work.

Chapter 9

Ifs



Quick View

Keywords

```
if then
if then else
end if
exit
```

Operators

```
=
is
>
<
>=
<=
<>
```

Operators *cont'd*

```
is within
and
or
not
```

Concepts

`if` statements (decision-making)
Control structures
Boolean expressions
Relational operators
Logical operators
Nested `if` statements

Chapter 9

Ifs

By strict definition, computers are not intelligent. Although they excel at following a series of instructions rapidly and accurately, any intelligence that computers appear to exhibit comes from their programming. Probably the single most important feature for making a program act intelligently is the power to make choices. HyperTalk provides this capability through the `if` family of statements. We explore them in this chapter. These statements make choices by making evaluations: Is the mouse button down? Is the global variable `it` less than 10? To understand decision-making, we also should know the logical bases for these evaluations. That, too, we investigate in this chapter.

You'll find `if` statements very useful. As we began to see in chapter 8, "Keyboard," you can use `if` statements to process responses to `ask` or `answer` commands. For example, if a user selects a Cancel button in a dialog box, you can use an `if` statement to terminate the script then and there, or you can have a script behave differently depending on the location of the mouse or the state of a key.

if then Statements in HyperTalk—An Overview

The simplest form of decision-making statement in HyperTalk has the form shown in Figure 9-1.

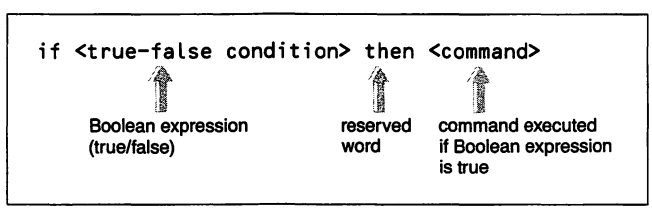


Figure 9-1. The `if then` statement syntax.

This statement has three elements. First, the keywords `if` and `then` provide a framework or structure for the command, separating a test condition from a pos-

sible action. Second, the `<true-false expression>` is a test condition that determines whether the following action is taken. Finally, the `<command>` represents the action to be performed if the test condition is true. If the `<true-false expression>` is false, the `<command>` part is skipped. Because the `if then` statement controls which commands are executed, it often is called a *control statement* or *control structure.*

A simple example of the `if then` statement is `if the shiftKey is down then beep`. The `<true-false expression>` is `the mouse is down`, and the `<command>` is `beep`. The function `the shiftKey`, recall, returns `down` or `up` depending on whether the Shift key is down or up, so the test condition compares the value returned by `the shiftKey` function with the HyperTalk constant `down`. If the two values are the same, the test condition is true and the beep command is executed.

As you can see, the `if then` statement provides a straightforward method for decision making: if A is true, do B; otherwise, don't. This is an example of "two-valued" logic. Test conditions can have one of just two possible values: `true` or `false`. HyperTalk doesn't have to deal with the more perplexing values we humans face, such as "maybe" or "mostly true."

Let's take one more look at the expressions in an `if then` statement. The `<command>` can be any of the commands we've used so far, for example, `put "Duck!" into the message box` or `answer "Do you still eat diskettes?" with "Yes" or "No"`.

The `<true-false expression>` is a new form that we study in this chapter. Typical examples are `the shiftKey is down` and `the mouseH > 256`. The second expression is true if the horizontal position of the mouse (`the mouseH`) is greater than 256 and is false otherwise. The expression also can be a function, such as `the mouseClick`, that returns values of `true` or `false`. We'll run through the full range of these expressions soon.

As these examples suggest, all those mouse and keyboard functions you studied in the last two chapters are used frequently with `if then` statements so that you can use the mouse and keyboard to control which actions are performed.

There are variations of these basic `if then` forms. To use any variations, you first need to know how to devise `<true-false expressions>`, so that's the first topic we take up in detail.

true false Expressions (Boolean Expressions)

Expressions are combinations of HyperTalk elements, such as constants, operators, containers, properties, and functions, that have a value. For example, the construction 4 + 2 * 3 has the value 10, so it is an expression, as is `random(10) * (the number of cards)`. More particularly, these are arithmetic expressions because you get numbers when you evaluate them. With `if then` statements, however, we use expressions that evaluate to true or false, not to numbers. So they are called *true-false expressions*. To see what we mean, try the following:

Example 1:

1. Go to your HyperTalk Bible stack and open the Message box.

2. Enter

```
2 = 2
```

Result:

The Message box displays the word `true`.

HyperTalk interprets the expression 2 = 2 as asking the question, "Is 2 equal to 2?" Because 2, indeed, is equal to 2, HyperTalk says the value of this expression is `true`, and it then replaces the expression in the Message box with its value.

3. Now enter the expression

```
3 = 2
```

Result:

The Message box displays `false`.

This time HyperTalk finds the assertation that 3 equals 2 is false, so it replaces the expression with its value: `false`.

Boolean Expressions

True-false expressions are called Boolean expressions after George Boole, the nineteenth century mathematician who showed how to represent statements and problems of logic in the symbolic language of mathematics, thus rendering logic (at least, two-valued logic) to a mathematical subject. Similarly, `true` and `false` are Boolean constants.

If we could use constants only in Boolean expressions, as in 3 + 4 = 8, they would not be of much interest, but HyperTalk lets us use containers, properties, and functions, too.

Example 2:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Enter

```
get the number of cards
```

3. Enter

```
it = 1001
```

Result:

The Message box displays `false`—unless you just happen to have 1001 cards! Step 2 obtains the number of cards in your HyperTalk Bible stack and places them in the global variable `it`. Step 3 compares the value of the variable to 1001. HyperTalk then replaces the expression with its value: `true` (unlikely) or `false`. This is an example of comparing a variable with a constant. You also can compare a property:

4. Enter

```
the number of cards = 1001
```

Result:

Again (most likely), the Message box displays `false`.

Finally, let's try using a function:

5. Enter

```
the mouse = up
```

Result:

The Message box displays `true`.

Here the value returned by the function `the mouse` is compared with `up`. Notice that in HyperTalk, Boolean expressions are not limited to explicit numbers; they can use HyperTalk constants like `up`, `down`, and `empty`. Also, Boolean expressions can compare strings, for instance, comparing the response of an `ask` command to the string `"Yes"`.

Boolean expressions, then, can involve various kinds of values, such as numbers, strings, and HyperTalk constants. The value of the Boolean expression, however, is restricted to being `true` or `false`.

HyperTalk has two main families of Boolean expressions: *relational expressions* and *logical expressions.* All our examples so far have been the first kind. Is A equal to B? Is C greater than D? The relationships concern equality and inequality. Logical expressions, on the other hand, use relational expressions as basic units. Are this relation *and* that relation both true? Is the mouse down *or* the Shift key down? Logical expressions allow you to construct more complex test conditions from simpler ones. We'll look at these two families of Boolean expressions next.

Relational Expressions

The first broad class of Boolean expressions consists of those that compare two values. Is this the same as that? If you look at a HyperTalk relational expression, such as `it = 1001`, you see that it has three parts. First (and third) are the two values being compared: `it` and 1001 in our example. Then, there is the *operator* used to make the comparison; in this example `=` is the operator. In chapter 5, "Simple Math," you worked with arithmetic operators such as `+` and `*`. An arithmetic operator combines two numbers into an arithmetic expression (such as 40 + 2) that has a numeric value (42). Similarly, HyperTalk uses several *relational operators* to construct relational expressions that have true-false values.

Hypertalk has operators to express six different possible relationships. We look at one relationship, `equality`, in detail to gain a general understanding of how these relational operators work. Then, we discuss the remaining five relationships.

Equality: = and is

You can use the operators `=` (the equal sign) and `is` interchangeably to test whether two values are the same. We've already tried 2 = 2 in the Message box. Here are some more examples to try in the Message box:

Example 3:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Enter

```
the shiftKey is down
```

into the message box

Result:

The Message box displays `false`.

3. Enter

```
2 + 5 = 25
```

Result:

The Message box displays `false`.

If you replace `is` with = in Step 2 and = with `is` in Step 3, the examples work the same. The choice between the two forms is mainly an aesthetic one. The `is` operator looks more natural when used with words, and the = operator looks more natural with numbers.

4. Enter

```
put 30 into age
```

5. Enter

```
age = 10
```

Result:

The Message Box displays `false`.

6. Enter

```
age is 30
```

Result:

The Message box displays `true`.

7. Enter

```
age + 15 = 50 / 2 + 20
```

Result:

The Message Box displays `true`.

Don't Use = for Assignment

Many languages, including BASIC and C, use = for assignment. In these languages `age = 10` means "Assign the value 10 to age." If you are used to that convention, *be careful.* In HyperTalk, `age = 10` always means comparison, never assignment. As Step 6 shows, age still has the value 30. To make assignments in HyperTalk, use `put`, as in Step 4.

This last example shows that you can use variables and arithmetic expressions on either side of the equality operator. HyperTalk evaluates each expression, and then compares the final values to decide whether the entire expression is true. HyperTalk uses the standard arithmetic precedences discussed in chapter 5, "Simple Math," to determine in which order to do the arithmetic, as shown in the following.

```
age + 15 = 50/2 + 20    ?      ←substitute values and evaluate expressions
  30 + 15 = 25    + 20    ?
        45 = 45              ?
          TRUE
```

These operators can be used with strings, too. Recall that a string is just a series of characters, such as `Hi ho!` or `Glorpf` or `2002i`. Card names, for example, are strings. HyperTalk uses double quotes, as in `"Mouse Card"` to indicate strings. In this instance, `Mouse Card` is the actual string; the quotes set it off as a string. Also, strings can be represented by containers. For instance, `put "Henry" into king` stores the string `Henry` in the variable `king`.

Example 4:

1. Go to your HyperTalk Bible stack and create a card called `If Then`.
2. Create a button called `Biffy` and put the following script into it:

```
on mouseUp -- Biffy button
  ask "What's your favorite name?"
  if it is "Biffy" then answer "Thanks!"
end mouseUp
```

3. Click the Biffy button and get a dialog box that asks `What's your favorite name?`
4. Enter `Biffy` in the dialog box.

Result:

An answer box appears with this message: `Thanks!`

When you enter `Biffy` in the dialog box, the string `Biffy` is assigned to the global variable `it`. The `if then` statement compares the contents of `it` with the string `Biffy`. The two are the same, so the `Thanks` message is given.

5. Clear the answer box, click the Biffy button again, and this time respond with `biffy`.

Result:

Again, the answer box appears with `Thanks!`

This should remind you of what you learned in chapter 1: HyperTalk does not distinguish between upper- and lowercase when it compares characters and strings. To HyperTalk, `Biffy` and `biffy` are the same. That means you don't have to check for all possible ways of typing an answer, such as `BIFFY`, `Biffy`, `biffy`, and `bIfFy`.

6. Repeat Step 5, except respond by entering `Biffy` and pressing the spacebar before striking the Return key.

Result:

This time there is no answer box thanking you.

This shows that HyperTalk counts a space as a character. Hence, `"Biffy "` (note the space after the `y`) contains 6 characters and does not match the five characters of `"Biffy"`.

Other Relational Operators

The other relational operators (see Figure 9-2 for syntax) are used in the same manner as the equality operator. That is, each is used with two values (*operands*), one to the left and one to the right.

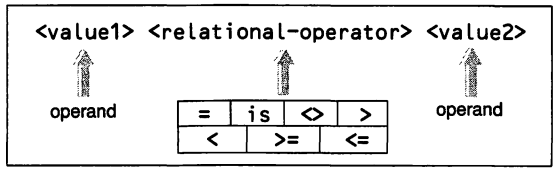


Figure 9-2. Relational operator syntax.

The result is a relational expression that has a value of `true` or `false` depending on the relationship between the two operands. Table 9-1 lists them all, including = and `is` for completeness.

Table 9-1. HyperTalk Relational Operators

Operator	Meaning
=	equals (is the same as)
is	same as =
<>	is not equal to
>	is greater than
<	is less than
>=	is greater than or equal to
<=	is less than or equal to
is in	first operand is in the second (for strings)
contains	first operand contains the second (for strings)
is within	first operand is a point within the second operand (a rectangle) (Version 1.2)

Briefly, here is what the new operators do. The <> operator returns `true` if two quantities are not equal. So `4 <> 5` is `true`. It's the logical opposite of =. The > operator returns `true` if the value to its left is greater than the value to its right. Thus, `9 > 4` is `true`. Similarly, the < returns `true` if the value to its left is less than the value to its right, making `9 < 4` evaluate to `false`. In each case, the open end of the operator faces the value that is being tested to be the larger. Next, the >= operator returns `true` if the value to its left is greater than or equal to the value to its right. So both `5 >= 4` and `5 >= 5` are true. Similarly, the operator <= produces a value of `true` if the value to its left is less than or equal to the value to its right. Note that >= and `not` > is the logical opposite of <.

To see how the other operators work, try typing into the Message box a few expressions using them:

Example 5:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Put the mouse near the middle of the card.
3. Enter

```
the mouseH > 500
```

Result:

The Message box displays `false`.

This shows that the horizontal mouse location is not greater than 500.

4. Enter

```
the mouseV < 300
```

Result:

The Message box displays `true`.

This shows that the vertical mouse position is less than 300.

5. Enter

```
put 5 into sides
```

6. Enter

```
sides >= 4
```

Result:

The Message box displays `true`.

7. Enter

```
sides <> 5
```

Result:

The Message box displays `false`.

8. Enter

```
sides <= 5
```

Result:

The Message box displays `true`.

The last three comparisons show that it's true that the value of sides is greater than or equal to 4, that it's false that the value is not 5, and that it's true that the value is less than or equal to 5.

Another Way to Say Greater Than

There are two other characters you can use in your scripts to represent the HyperCard operators >= and <=. These characters can be accessed normally by pressing 〉= or 〈= on the keyboard. You can also get them by pressing Option-〉 or Option-〈. The final characters look like this in a script: ≥ (greater than or equal to) and ≤ (less than or equal to).

You can use these operators with strings, too. In that case, `"less than"` means "precedes alphabetically."

Example 6:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Enter

```
"Wasco" > "New York"
```

Result:

The Message box displays `true`.

HyperTalk responds with `true` because W comes after N in the alphabet. Actually, Hypercard uses the ASCII code values for characters to extend the comparison to nonalphabetic characters.

is within

The `is within` operator is a Version 1.2 feature that takes two operands. The first is a point, and the second is a rectangle. A point is represented by the horizontal and vertical coordinates separated by a comma; for instance, 20,30 is a point 20 units from the left edge of the card and 30 units down from the top. A rectangle is represented by four numbers separated by commas. The first two numbers are the coordinates of the top-left corner, and the final two are the coordinates of the bottom-right corner. For instance, 5,10,40,50 represents the rectangle whose top left corner is 5 units from the left side of the card and 10 units from the top; the bottom right corner is 40 units from the left side of the card and 50 units from the top. The `is within` operator yields a result of `true` if the point is within the rectangle. Try it:

Example 7:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Enter

```
"20,30" is within "5,10,40,50"
```

in the Message box.

Result:

The Message box displays `true`.

Note that we use double quotes. This helps HyperTalk recognize 20,30 and 5,10,40,50 as being single entities (a point and a rectangle) instead of being a list of things.

You also can check to see whether the point is outside the rectangle by using `is not within`:

3. Enter

```
"20,30" is not within "5,10,40,50"
```

in the Message box.

Result:

The Message box displays `false`.

More typically, you would use this operator with functions or properties that provide points and rectangles. For example, the functions `the mouseLoc` and `the clickLoc` return points, as does the `location` property. The `rectangle` property returns a rectangle, and we use the `is within` operator in those chapters.

Using Boolean Expressions

What makes the `if` statement powerful is that it can be used to compare values that can change. For example, you can make an action dependent on the mouse position:

Example 8:

1. Go to the If Then card of your HyperTalk Bible stack.
2. Open the card script and enter the following text:

```
on mouseStillDown
  if the mouseH > 256 then put "Mouse too far to right!"
  if the mouseH < 256 then put "Mouse too far to left!"
  if the mouseH = 256 then put "Ahh! Just right!"
end mouseStillDown
```

3. Place the mouse to the far right of the card, press the mouse button, and while the button is still down, move the mouse back and forth across the card.

Result:

Initially, the message `Mouse too far to right!` appears in the Message box. Whenever you move the mouse to the left of midcard, the message changes to `Mouse too far to left!`, and it reverts to the first message when you move the mouse to the right of midcard. When you hit 256 exactly, you get the message `Ahhh! Just Right!`

Now you have a script whose behavior changes as the mouse position changes. Later, we'll see how to use logical operators to widen the mouse's comfort zone. (Here, to keep the lines short, we use the fact that `put` positions values in the Message box if we don't explicitly tell it to place them elsewhere.)

Typically, then, an `if` statement compares a function value or a variable to a constant or to another function value or variable. Here's another example:

Example 9:

1. Go to the If Then card of your HyperTalk Bible stack and create a new button called `Show and Hide`.
2. Put the following text into the button script:

```
on mouseEnter
  if the shiftKey is down then show the message box
  if the shiftKey is up then hide the message box
end mouseEnter
```

3. Open the Message box, and then move the mouse into the Show and Hide button.

Result:

The Message box disappears.

4. Move the mouse out of the button, hold the Shift key down, and while the key is still down, move the mouse back into the Show and Hide button.

Result:

The Message box reappears.

A `mouseEnter` message is sent to the Show and Hide button when the mouse enters the button. If the Shift key is up at the time the on mouseEnter handler is processed, the button script executes the `hide the message box` command. If the Shift key is down at the time the on mouseEnter handler is processed, the script executes the `show the message box` command.

One important point to note here is that it is the `mouseEnter` message that initiates the hiding and showing of the Message box; the Shift key is used to modify the effect of the `mouseEnter` message. Pressing and releasing the Shift key while the mouse is inside the button has no effect. What counts is the state of the Shift key just after the mouse crosses the boundary into the button. (The script doesn't actually check the status of the Shift key until after the `mouseEnter` message is sent.)

Is it possible that both actions could take place during a single `mouseEnter` event? That is, could the Shift key be up when the first `if then` statement is processed and then be down by the time the second `if then` statement is executed? In principle, the answer is yes. Because we have two `if then` statements, the status of the Shift key is checked twice, so you could get two different answers. If you press the Shift key at just the right moment, the script first detects the key as up and hides the Message box, and then detects that the key is down and shows the box again. When we discuss the `if then else` structure, you see how to avoid this kind of problem.

Let's look at one more example involving several `if then` statements. This one selects visual effects based on the states of the Shift, option, and command keys.

Example 10:

1. Go to the If Then card of your HyperTalk Bible stack and create a button called `This Card`.
2. Put the following script into the new button:

```
on mouseUp -- This Card button
  if the shiftKey is down then visual effect dissolve to grey
  if the optionKey is down then visual effect checkerboard to black
  if the commandKey is down then visual effect scroll right
  go to this card
end mouseUp
```

3. Hold the Shift key down and click the mouse in the This Card button.

Result:

You see a dissolve-to-grey visual effect.

4. Hold both the Shift and option keys down while clicking the mouse in the This Card button.

Result:

You see a dissolve-to-grey visual effect followed by a checkerboard-to-black visual effect.

Because there are three separate `if then` statements, each visual effect can be set independently of the others, and because visual effects are placed in a queue, all the effects you select are used.

Comparison Converts Strings to Numbers

Comparison operators always attempt to interpret strings as numbers. Thus, the expression "5" > "100" is false as expected, but the expression "*5" >*20" is true because the two strings are compared rather than the numbers. If you want to use the `compare` operators to compare actual strings rather than numbers, a nonnumeric character must be appended to each of the values to force HyperTalk to interpret them as strings. Your expressions appear as

```
if "%" & var 1 > "%" & var2 then . . .
```

Also keep in mind that the equality comparisons = and <> use only the primary ordering of characters, so upper- and lowercase and the special Macintosh diacritical characters are ignored.

If you have Version 1.2 of HyperCard, you may have trouble setting the checkerboard and scroll-right effects simultaneously because HyperCard interprets the option-key and Control-key combination as a command to show where the buttons are. You can get around this, however, by placing the mouse in the button, holding the mouse button down, pressing the option key and the Control key, releasing the mouse button, and then releasing the keys. Having the mouse button down disables the HyperCard intervention, and the script investigates the key states on the `mouseUp` message.

Logical Expressions

You may want to test for two or more conditions simultaneously. For example, suppose that you want to see whether the horizontal location of the mouse is within six pixels of card center. That means you want the horizontal position to be more than 250 and less than 262. HyperTalk lets you use logical operators to combine two or more Boolean expressions into a single expression or to negate an expression. The operators are called `and`, `or`, and `not`. Let's look at them.

The and *Operator*

The `and` operator, the syntax of which is shown in Figure 9-3, joins two Boolean expressions into one Boolean expression that is true only if both of the original expressions are true.

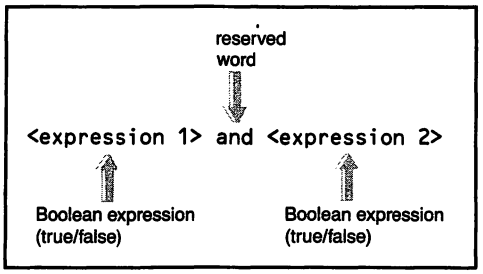


Figure 9-3. The and operator syntax.

Let's modify Example 8 to widen the mouse's comfort zone:

Example 11:

1. Go to the If Then card of your HyperTalk Bible stack and open the card script.
2. Modify it to read as follows:

```
on mouseStillDown
  if the mouseH >= 262 then put "Mouse too far to right!"
  if the mouseH <= 250 then put "Mouse too far to left!"
  if the mouseH > 250 and the mouseH < 262 then put "Ahh! Just right!"
end mouseStillDown
```

3. Move the mouse around the card while the mouse button is down.

Result:

A message in the Message box reflects the mouse's position.

In Example 8, you had to place the mouse exactly at horizontal location 256 to satisfy the mouse; now, anything in the range 251–261 works. The key point is that the and statement lets us test for two conditions simultaneously; in this case, that allows you to test for a range of values rather than for a single value.

In addition to allowing you to test for ranges, the and operator lets you check to see whether two unrelated conditions hold simultaneously.

Example 12:

1. Go to the If Then card of your HyperTalk Bible stack and create a new button called Make Me Beep.
2. Put the following script into the new button:

```
on mouseUp -- Make Me Beep button
  if the shiftKey is down and the optionKey is down then beep 3
end mouseUp
```

3. Hold the Shift key down and click the mouse in the Make Me Beep button.

Result:

Nothing happens.

4. Hold the option key down and click the mouse in the new button.

Result:

Nothing happens.

5. Hold both the option key and the Shift key down and click the mouse in the new button.

Result:

Three beeps.

This shows very clearly that the `and` operator requires both conditions to be true in order for the combined expression to be true.

The or *Operator*

The `or` operator (see Figure 9-4 for syntax) also joins two Boolean statements into one. This time the resulting expression is true if either (or both) of the original expressions is true.

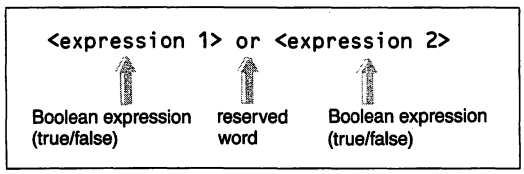


Figure 9-4. The `or` operator syntax.

Let's look at a simple example.

Example 13:

1. Go to the If Then card of your HyperTalk Bible stack.
2. Clone a copy of the Make Me Beep button. (Select the button tool, click the button, hold the option key down, and drag off the copy.) Call it `Make Me Beep, 2`.
3. Give it the following script.

```
on mouseUp -- Make Me Beep,2 button
  if the shiftKey is down or the optionKey is down then beep 3
end mouseUp
```

4. Hold the Shift key down and click the mouse in the Make Me Beep button.

Result:

Three beeps.

5. Hold the option key down and click the mouse in the new button.

Result:

Three beeps.

6. Hold both the option key and the Shift key down and click the mouse in the new button.

Result:

Three beeps.

So if either button is down, the script executes the `beep 3` command. Even if both are down, the script is executed. (This form of `or` is called an *inclusive or* because it includes the case of both expressions being true.) Logic (but not HyperTalk) also has an *exclusive or*, which requires that one or the other expression, but not both, be true.

Let's look at an example that uses both `and` and `or`. It creates visual effects that depend on where the mouse enters the field.

Example 14:

1. Go to the If Then card of your HyperTalk Bible stack and create a new field called `Zoom`.
2. Leave the field at the card center and choose rectangle style so that it will be easy to find.
3. Place the following text in the field script:

```
on mouseEnter
  if the mouseH < 256 and the mouseV < 171 then visual effect¬
  zoom in to grey
  if the mouseH >= 256 or the shiftKey is down then visual effect¬
  iris open to black
  go to this card
end mouseEnter
```

4. Move the mouse into the field from the upper-left corner.

Result:

The zoom-in-to-grey visual effect is executed.

In this case, the test condition for the first `if then` statement is met, because the left corner of the field has x 〈 256 and y 〈 171, assuming you didn't move the field.

5. Move the mouse from outside the field into the field from the lower-left corner.

Result:

Nothing happens.

This time, neither test condition was met, so the `go to this card` command was executed without visual effects.

6. Hold the Shift key down and move the mouse from outside the field into the field from the lower-left corner.

Result:

The iris-open-to-black visual effect is executed.

This time, the Shift key was down (meeting the conditions for the second `if then` statement), so the iris effect was executed.

7. Move the mouse from outside the field into the right half of the field.

Result:

The iris-open-to-black visual effect is executed.

Moving the mouse into the right half of the field satisfies the mouseH 〉= 256 condition, so, again, the iris effect is executed.

Note that the `and` operator is more restrictive than `or`. The condition `the mouseH < 256` restricts the entry point to the left half of the card, and the condition

`the mouseV < 171` restricts the entry point to the upper half of the card. Between the two, they restrict the entry point to just the upper-left quadrant of the card. If you use `or` rather than `and` for the first `if then` statement, the effect is to expand the allowed entry points rather than to restrict them further. See Figure 9-5.

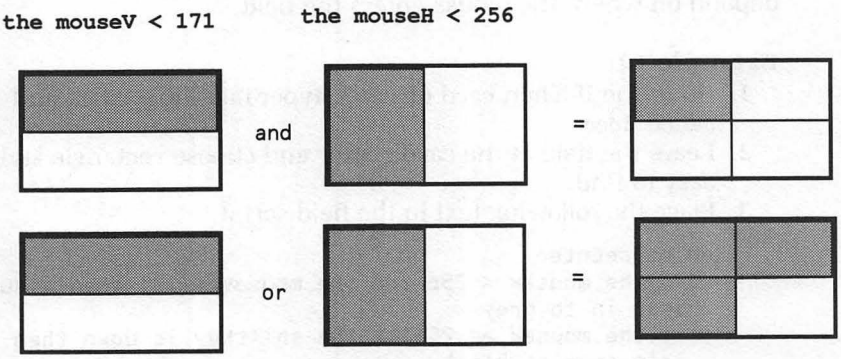


Figure 9-5. `and` versus `or` Operators.

Using `or` in the second `if then` statement gives us two independent means of producing the iris effect. Again, `or` expands the possibilities.

or Is the More Inclusive Operator

Ordinarily, we think of `and` as being more inclusive than `or`. For instance, would you rather have $200,000 and a new house or would you prefer $200,000 or a new house? But in logic, as we've seen, `or` is the more inclusive operator.

The not *Operator*

The `not` operator negates a Boolean expression. That is, it turns a false expression to true and vice versa. Figure 9-6 shows the syntax for the `not` operator.

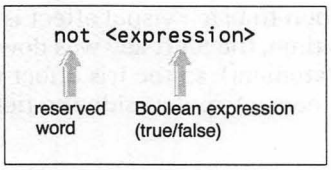


Figure 9-6. The `not` operator syntax.

Example 15:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Enter

```
not true
```

Result:

The Message box displays `false`.

3. Enter

```
not 10 > 4
```

Result:

HyperTalk says it `Can't understand not`.

The problem here is that `not` is supposed to be applied to true-false expressions. But HyperTalk gives `not` a higher *precedence* than `>`, so this expression is interpreted as `(not 10) > 4`. That is, HyperTalk tries to apply `not` to the value `10` rather than to the expression `10 > 4`. The solution is to use parentheses to show what you really mean. (Table 7-1 in Chapter 7 lists HyperTalk operator precedences.)

4. Enter

```
not (10 > 4)
```

Result:

The Message box displays `false`, because `10 > 4` is true.

When applying `not` to an expression, therefore, you should place the whole expression in parentheses. Then, HyperTalk will know what you mean.

5. Enter

```
not (the mouse is down)
```

Result:

The Message box displays `true`.

Because the mouse is up, the expression `the mouse is down` is false, and `not false` is true.

6. Enter

```
the mouse is not down
```

Result:

The Message box displays `true`.

7. Enter

```
not down
```

Result:

HyperTalk says it `Can't understand not`.

The last two steps show an interesting point: HyperTalk knows what you mean by `is not`. If you have an expression of the form `A is not B`, HyperTalk treats it as if you said `not (A is B)`. Just looking at step 6, you might think that the `not` operator is applied to down and that not down is up. But step 7 shows that is not so; step 6 really is the same as step 5.

Note that usually you can avoid using `not` by revising the expression. For instance, `not (the mouseH > 200)` can be rewritten as the `mouseH <= 200`, and `the mouse is not down` can be replaced with `the mouse is up`. Normally, the `not`-free form is easier to understand.

Precedence for and, or, *and* not

Let's return to the topic of operator precedence. If you want to use more than one of the logical operators in the same expression, you should be aware that operator precedence can be important.

To see the practical consequences of operator precedence, you need to look at logical expressions that are more involved than what you've seen so far. For example, suppose that you have these design requirements for a field script:

1. Moving the mouse from outside the field into the upper-left quadrant of the field triggers a zoom-in visual effect.
2. Moving the mouse into the right half of the card triggers an iris-open effect if the mouse also is in the bottom half of the card or if the Shift key is down.

A *quadrant* is one of four equal portions of the field produced by dividing the field in half horizontally and vertically. Figure 9-7 illustrates the effects desired for each quadrant.

In Example 14, we developed the code for the first requirement: `if the mouseH < 256 and the mouseV < 171 then visual effect zoom in to grey`. The first part of the second requirement is that `the mouseH >= 256`, and the second part is that `the mouseV >= 171 or the shiftKey is down`. This suggests that you express the entire second requirement as follows: `the mouseH >= 256 and the mouseV >= 171 or the shiftKey is down`. Unfortunately, this is wrong, but let's try it first to see where the problem is.

Example 16:

1. Go to the If Then card of your HyperTalk Bible stack and open the script for the field Zoom.
2. Give it the following text.

```
on mouseEnter
  if the mouseH < 256 and the mouseV < 171 then visual effect¬
  zoom in to grey
  if the mouseH >= 256 and the mouseV >= 171 ¬
  or the shiftKey is down then visual effect iris open to black
  go to this card
end mouseEnter
```

3. Move the mouse from outside the field into the upper-left quadrant.

Result:

You see the zoom-in effect, meeting requirement 1.

4. Move the mouse from outside the field into the lower-left quadrant.

Result:

You see the iris-open effect, meeting requirement 2.

5. Move the mouse from outside the field into the upper-right quadrant.

Result:

You see no effect, meeting requirement 2.

6. Repeat Step 5, but with the Shift key down.

Result:

You see the iris-open effect, meeting requirement 2.

Well, so far, the program is working just as planned, but, being honest, we have to show one more test.

7. With the Shift key down, move the mouse from outside the field into the lower-left quadrant.

Result:

You see the iris-open effect.

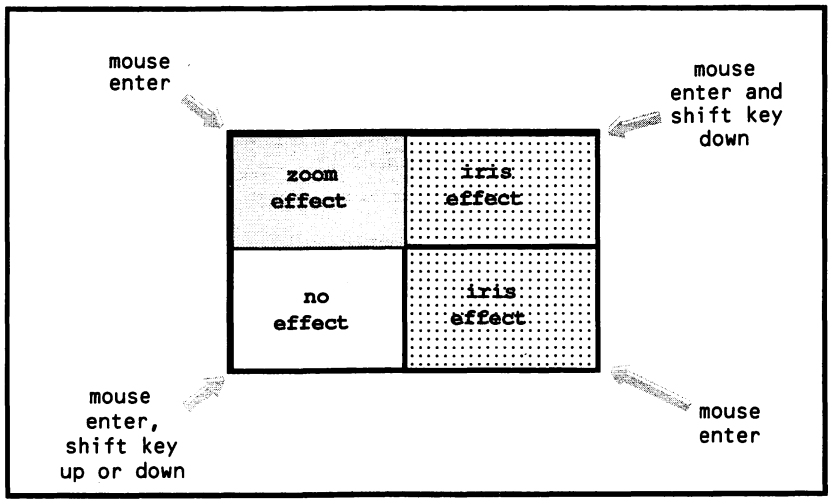


Figure 9-7. Desired behavior.

The correct response is no visual effects, so our script has failed. The reason is that `and` has a higher precedence than `or`. We wanted the tests to be grouped as shown in Figure 9-8.

```
if the mouseH < 256 and the mouseV < 171 or the shiftKey is down
```

Figure 9-8. Desired grouping.

However, because `and` has higher precedence than `or`, `and` is applied first and groups the first two tests together, as shown in Figure 9-9.

```
if the mouseH < 256 and the mouseV < 171 or the shiftKey is down
```

Figure 9-9. Actual grouping.

As a result, having the Shift key down on field entry always results in the iris-open effect. Indeed, if the mouse enters the upper-left quadrant with the Shift key down, you get both effects: the entry quadrant selects the zoom-in effect, and the Shift key selects the iris effect. The fix is to use parentheses to force the desired grouping.

Example 17:

1. Return to the script for the field Zoom and add parentheses as shown:

```
on mouseEnter
  if the mouseH < 256 and the mouseV < 171 then visual effect¬
  zoom in to grey
  if the mouseH >= 256 and (the mouseV >= 171 ¬
  or the shiftKey is down) then visual effect iris open to black
  go to this card
end mouseEnter
```

2. Repeat Steps 3–6 of Example 16.

Result:

The script acts as before—correctly.

3. Now, with the Shift key down, move the mouse from outside the field into the lower-left quadrant.

Result:

Nothing happens, which is the correct behavior.

Parentheses save the day. If you're not sure of the precedence order, you can check the help stack or add parentheses to make your desired order clear. Then, you don't have to worry whether they really are necessary.

The *if* Statements

Now that we've seen how to put Boolean expressions together and what they mean, let's return to the HyperTalk `if` statements and see how they can add to the power of your scripts. The `if` statement comes in several variations. All start by using a Boolean test condition, and we've covered that aspect pretty well. The variations add two useful features. The first is the capability of having an `if then` statement execute several commands rather than just one. The second is the capability of specifying one action for a true response and a second for a false response. Before going into these features, let's review the basic `if then` form.

The *if* Statement

You've already seen the syntax for the basic `if then` statement, along with many examples of how the statement works. More formally, control statements often are il-

lustrated using *flow charts*. Such charts show the flow of program control. Figure 9-10 shows such a chart for the `if then` statement.

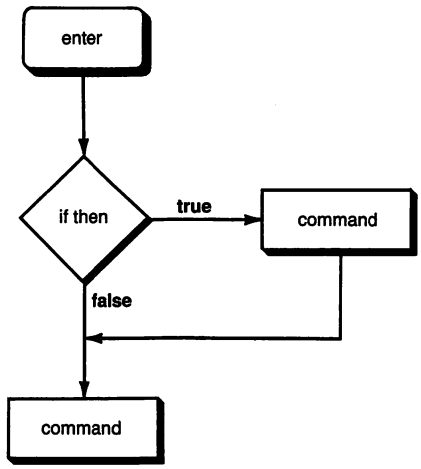


Figure 9-10. `if then` flow chart.

This chart shows that an `if then` statement is a *branching* statement, one that provides two possible branches of statements for the program to follow. If the test condition is true, the program follows the branch to the right. Otherwise, program flow continues down the lower branch. In this particular case, the true branch rejoins the main flow before the next command along that branch.

Here's one more example of the basic form:

Example 18:

1. Go to the If Then card of your HyperTalk Bible stack and create a button called `Single If`.
2. Enter the following script in the button.

```
on mouseUp -- Single If button
  put empty into the message box
  answer "show the time?" with "Yes" or "No"
  if it is "Yes" then put the time into the message box
end mouseUp
```

3. Click the Single If button.

Result:

You get a dialog box on the screen and an empty Message box.

4. Click the Yes response.

Result:

The Message box displays the time.

The `if` statement provides a natural tool for processing the responses to a dialog box; with it, we can make a particular response select a particular course of action.

Long Lines and an Alternative Syntax

If your `if` statement gets too long, you can use option-return (which displays as ¬) to continue the line on the next line. We did this in Example 17:

```
if the mouseH < 256 and the mouseV < 171 then visual effect¬
zoom in to grey
```

Although a line divided in this manner takes two lines on the screen, it counts logically as one line.

Another option is to use the alternative syntax shown in Figure 9-11.

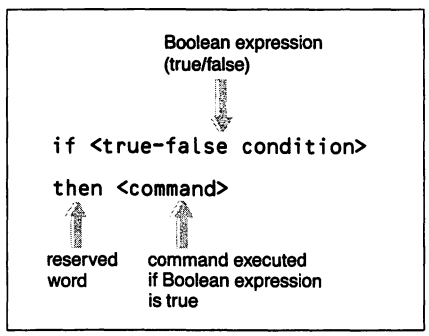


Figure 9-11. The alternative `if then` statement syntax.

The important points in Figure 9-11 are that this form is two lines long and that the second line begins with `then`. Here is the previous example written with this syntax:

```
if the mouseH < 256 and the mouseV < 171
  then visual effect zoom in to grey
```

If you still need more room, you can use the option-return technique to extend either line.

Choosing Multiple Commands: *end if*

One limitation of the basic `if then` form is that it doesn't allow you to select more than one command to be executed if the test condition is true. You may want more,

however. For instance, Example 18 shows the time in the Message box if the user selects Yes. Suppose we want to accompany the display with a sound effect, something Big Ben-like. We can do so by using `end if`, the syntax of which is shown in Figure 9-12.

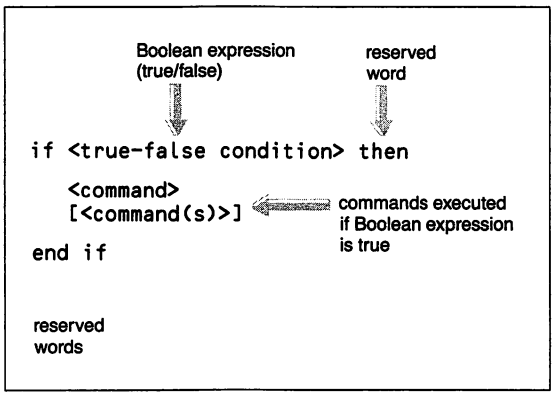


Figure 9-12. The `if end if` statement syntax.

All the commands between the `if then` and the `end if` are executed if the Boolean expression is true. Each command should be a valid HyperTalk command, and you can still use Option-return to extend individual command lines. Note that with this syntax, the `then` comes at the end of the first line and not at the beginning of the second.

Example 19:

1. Go to the If Then card of your HyperTalk Bible stack and create a button called `If end if`.
2. Give the button the following script:

```
on mouseUp -- If end if button
  put empty into the message box
  answer "Do you want to know the time?" with "Yes" or "No"
  if it is "Yes" then
    play "boing" "68 64 66 60 60 66 68 64"
    put the time into the message box
  end if
end mouseUp
```

3. Click the button, and then click Yes in the dialog box

Result:

First the dialog box appears. When you click Yes, the Macintosh sounds part of Big Ben's traditional tune, and the Message box displays the time.

The group of commands between the `if then` line and the `end if` line sometimes is called a *block*. This block has just two commands in it, but you can add more, or you can use just one command, if you like.

When you enter this script, note how the HyperCard editor automatically indents the block of commands between the `if` and the `end if`. After you enter the `end if` line, it is a good idea to press the Tab key. When you are in the script editor, pressing this key forces automatic indentation and alignment. If the script doesn't align the way you expect it to, chances are you've made a syntax error. For instance, if you leave the `then` out, you don't get indentation because HyperCard doesn't detect an `if then` statement.

The *if then else* Statement

So far our scripts have taken special action only when the test command has been true. The next step in `if` complexity lets you specify one action for the true case and a different action for the false case. See Figure 9-13.

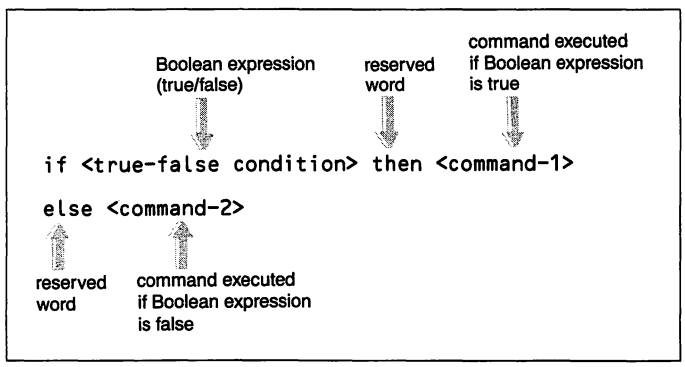


Figure 9-13. The `if then else` statement syntax.

If the `<true-false expression>` is true, the command following `then` is executed, and if the expression is false, the command following the `else` is executed. Figure 9-14 shows the flow chart for this form.

For instance, let's give the user a choice between a beep and a harpischord sound:

Example 20:

1. Go to the If Then card of your HyperTalk Bible stack and create a new button called `If Else`.
2. Put the following script into the button:

```
on mouseUp -- If Else button
  answer "Would like to hear a beep?" with "Yes" or "No"
  if it is "Yes" then beep
  else play "harpsichord"
end mouseUp
```

3. Click the button and select Yes in the dialog box.

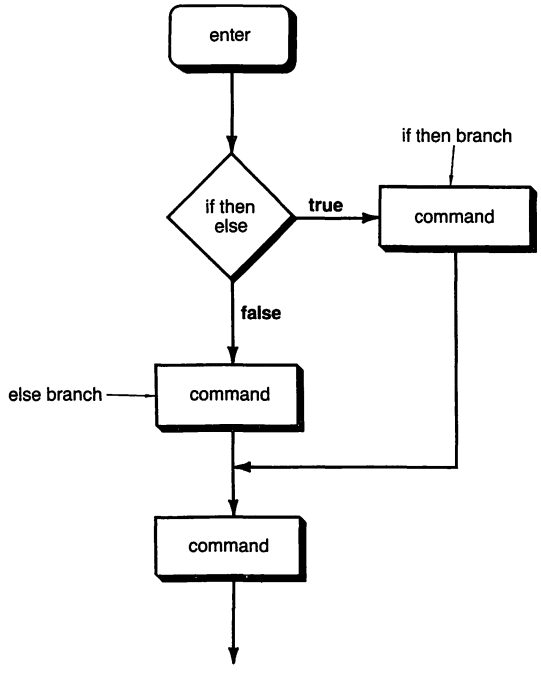


Figure 9-14. `if then else` flow chart.

Result:

You hear a beep.

4. Click the button and select No in the dialog box.

Result:

You hear the harpsichord sound.

We could have achieved the same effect by using two `if` statements:

```
if it is true then beep
if it is false then play "harpsichord"
```

However, this requires that HyperTalk make two comparisons: `it is true` and `it is false`. With the `else` form, HyperTalk only has to make one comparison, which saves time.

In this case, using the `else` form is a convenience and a time-saver. Sometimes you need the form to guarantee that one answer excludes the other. For example, in Example 9, we had this handler:

```
on mouseEnter
  if the shiftKey is down then show the message box
```

continued

```
  if the shiftKey is up then hide the message box
end mouseEnter
```

As we mentioned at the time, it's possible in principle for both commands to be selected because the state of the Shift key can change between the first and second `if then` statement. That problem disappears when you use the `if else` form:

```
on mouseEnter
  if the shiftKey is down then show the message box
  else hide the message box
end mouseEnter
```

This script tests the state of the Shift key just once, and that state determines which of the two possible actions is taken. There's no possibility of both actions being taken.

Variations of the *if else* Form

Just as with the simple `if else` structure, you can increase the number of commands for each choice by using blocks of commands. If you want more than one command executed for the true choice, you can use `if then` to mark the beginning of the block of commands and `else` to mark the end, as shown in Figure 9-15.

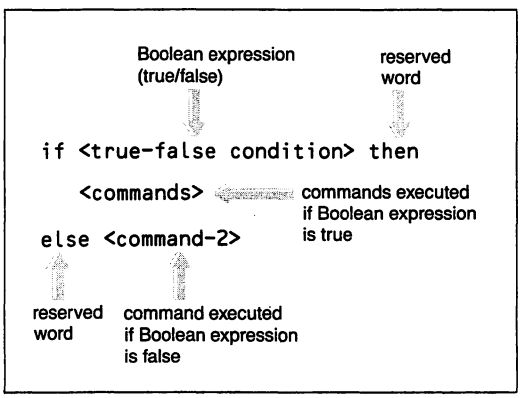


Figure 9-15. The `if then else` statement syntax, multiple commands.

You have to use `end if` if you want more than one command associated with the `else`. See Figure 9-16.

Let's use this last form `if then else` to create a paradoxical button.

Example 21:

1. Go to the If Then card of your HyperTalk Bible stack and create a button named `A truth follows`.

2. Size the button so that the whole name shows.
3. Give the following script to the button:

```
on mouseUp -- A truth follows button
  if the short name of me is "A truth follows" then
    set the name of me to "A lie follows"
  else
    set the name of me to "A truth follows"
  end if
end mouseUp
```

4. Click the button a few times.

Result:

The name changes from `A truth follows` to `A lie follows`, then back, and so on.

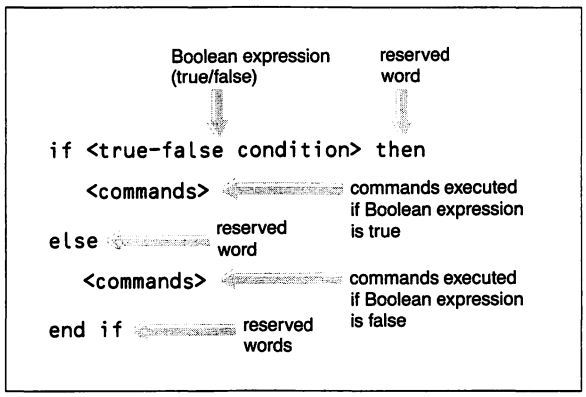


Figure 9-16. The `if then else end if` statement syntax.

Here, `me` is the button, and the short name of `me` is the button name without frills. If the name is `A truth follows`, the first part of the `if` statement sets the name to `A lie follows`. If the name is `A lie follows`, the test condition is false, and the `else` part is executed. Incidentally, if we placed `set the name of me to "A truth follows"` on the same line as the `else`, we could omit the `end if`.

Note that we don't ever have to test whether the name is `A lie follows` because we know that if the name is not `A truth follows`, the only possibility left is `A lie follows`. This illustrates the efficiency of the `else` variation.

For those of you with Version 1.2 or later, here's an example using the `is within` operator.

Example 22:

1. Go to the If Then card of your HyperTalk Bible stack.
2. Open the card script and add this text:

```
on mouseUp
  if the clickLoc is within "0,20,511,25" then hide menuBar
  else show menuBar
end mouseUp
```

3. Move the browse cursor to just under the menu bar and click the mouse.

Result:

The menu bar vanishes.

4. Move the browse cursor down the card a bit and click the mouse.

Result:

The menu bar vanishes.

The rectangle 0,20,511,25 defines a narrow bar 6 pixels tall running the full width of the card just below the Menu bar. By clicking inside and outside this triangle, you can hide and show the Menu bar without having to type commands in the Message box.

The *if then else if then else* Sequence

You can keep adding alternatives to an `if` structure by making the `else` command another `if` command. See Figure 9-17.

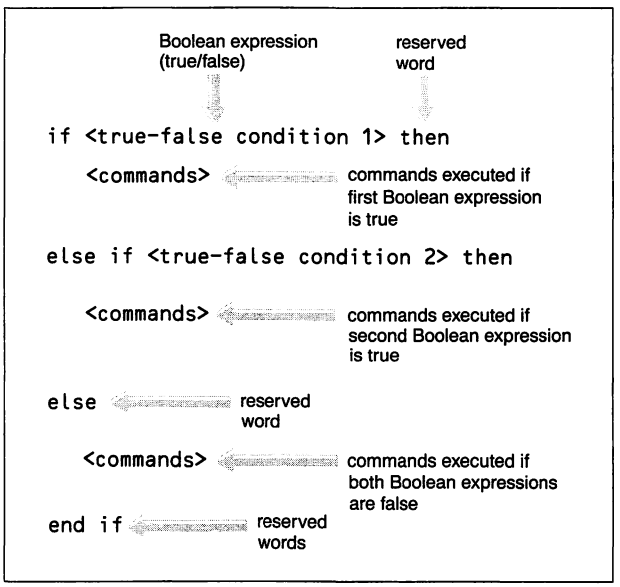


Figure 9-17. The `if then else if then else` statement syntax.

Here's a simple example that shows how to set up this sort of structure.

Example 23:

1. Go to the If Then card of your HyperTalk Bible stack and create a button called `How Many Boings`.
2. Give the button the following script:

```
on mouseUp -- How Many Boings button
  answer "How many boings?" with "one" or "two" or "three"
  if it is "one" then
    play "boing"" tempo 200 g
  else if it is "two" then
    play "boing" tempo 200 g c
  else
    play "boing" tempo 200 g c g
  end if
end mouseUp
```

3. Click the button.

Result:

A dialog box asks you how many boings you would like.

4. Contain your excitement at this request and click one of the choices.

Result:

The Macintosh boings the number of times you specified.

This dialog box offered three choices, and the `if` structure matches it perfectly by processing the three choices. Again, note that you don't have to test for all three choices because if a choice is not one of the first two, it must be the third.

By the way, because each of our command choices was a single command, we could have written the script this way:

```
on mouseUp
  answer "How many boings?" with "one" or "two" or "three"
  if it is "one" then play "boing" tempo 200 g
  else if it is "two" then play "boing" tempo 200 g c
  else play "boing" tempo 200 g c g
end mouseUp
```

This certainly is more compact. However, the other form is easier to read and easier to modify, especially if you wish to insert additional commands.

Nested *ifs*

As you've just seen, the `<command>` part of an `if then` statement can be another `if then` statement. We strung them together in a pattern suitable for processing a list of possibilities. Sometimes, however, you need to embed an `if` statement in the middle of another one. That is called *nesting* the statements. One common occasion for doing this is when you ask a question based on the response to the preceding ques-

tion. The user's first answer affects his choice of answers for the next question, leading to an `if` within an `if`. Here is an example.

Example 24:

1. Go to the If Then card of your HyperTalk Bible stack and create a button called `Nested Ifs`.
2. Place the following script in the button:

```
on mouseUp -- Nested Ifs
  answer "Which sound do you prefer?" with "Beeps" or "Boings"¬
  or "Dial Tones"
  if it is "Beeps" then                  -- outer if begins
    answer "How many Beeps?" with "1" or "2"
    if it is "1" then beep               -- 1st inner if
    else beep 2
  else if it is "Boings" then            -- 2nd part of outer if
    answer "Me, too"
    play "boing"  else                   -- 3rd part of outer if
    ask "Which tone? (0-9)" with "4"
    if it is empty then exit mouseUp  -- 2nd inner if, has a
    else if it > 9 then put 9 into it      -- new command (exit)
    else if it < 0 then put 0 into
    dial it
  end if                                -- outer if ends
end mouseUp
```

3. Click the button.

Result:

You see a dialog box. (see Figure 9-18).

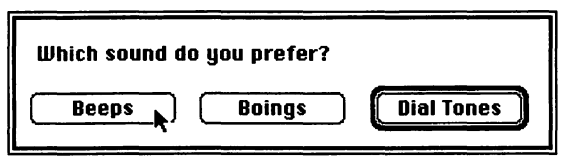


Figure 9-18. First dialog box.

4. Click Beeps.

Result:

You see a second dialog box (see Figure 9-19).

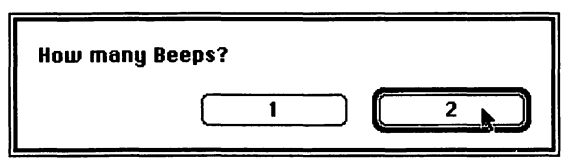


Figure 9-19. Second dialog box.

5. Click 2.

Result:

The Macintosh beeps twice.

Also try clicking choices 1 and 3 and see what happens.

The "outer" `if` statement is a three-part `if then else if then else` statement that handles the three choices to the original dialog box. The first "inner" `if` statement processes the two choices to the follow-up dialog box for the Beep choice. This is a nested statement. The second inner `if then` statement processes the follow-up dialog box to the dial tones selection. It, too, is a nested statement. Also note that we don't have to explicitly test for the Dial Tones response because, by elimination, that must be the response if it's not beep or boing.

Another thing to note is that the secondary dialog boxes assign new values to the global variable `it`. If you need to save the original response, you should put it into some other container.

This example also introduces two new topics: data checking and the `exit` command. Let's take a quick look at them.

Data Checking

The second nested statement illustrates a common use for `if` statements: data checking. This particular example requests a number in the range 0–9. If the user oversteps these bounds, the `if` statement resets the variable `it` to one in the requested range. In general, you have several possible responses to unacceptable input. You can substitute a default value, as we did here, you can use the `exit` command to terminate the script, or you can ask the user to try again. This is done most easily by using a repeat loop, as discussed in the next chapter. Also, using HyperTalk's string facilities (see chapter 11, "Strings") makes it possible to examine input more closely for acceptability—for example, discriminating between numeric and nonnumeric input.

The *exit handler* Command

The last example also makes use of one form of the `exit` command. Suppose that you are in the middle of a script and find that you want to quit the script right there and skip the rest. In Example 24, for example, the `ask` command puts up a dialog box with a Cancel button. If the user selects that button, the script should terminate. Well, selecting Cancel causes the variable `it` to wind up with the HyperTalk value `empty`. If the script detects that response, it executes the `exit mouseUp` command. As you would guess from its wording, it causes HyperTalk to exit the mouseUp handler right there, skipping the rest of the script. The command has a straightforward syntax, as shown in Figure 9-20.

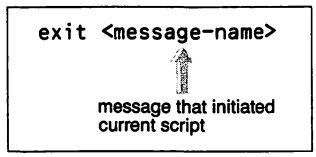


Figure 9-20. The `exit` command syntax.

Here `<message-name>` is the name of the message being managed by the handler containing the command. Typically, this command is used in conjuction with an `if` statement, as in our last example.

***exit to HyperCard* Savior**

If you are in a script and don't know how many levels you are deep in calls to other handlers that have called you, the statement `exit to HyperCard` pops you out of all submessages. All pending handlers terminate.

Let's look at another example that uses data checking and the exit command.

Example 25:

1. Go to the If Then card of your HyperTalk Bible stack and create a new button called `Square Root`.
2. Put the following script into the button:

```
on mouseUp -- Square Root button
  ask "Square root of what?"
  if it is empty then
    exit mouseUp
  else if it < 0 then
    answer "I don't do negative numbers!"
    exit mouseUp
  else
    answer sqrt(it)
  end if
end mouseUp
```

3. Click the button and respond to the dialog box by entering `36`.

Result:

An answer box displays `6`.

4. Click the button and click Cancel.

Result:

The script terminates.

5. Click the button and respond to the dialog box with `-16`.

Result:

An answer box displays `I don't do negative numbers!`

Here, you checked for no input and for the input of a negative number. The `sqrt()` function is one of HyperTalk's built-in functions (see chapter 13, "Math Functions"), and it finds negative numbers distasteful.

Summary Script

Let's look at one final example. This time, we use several `if then else` statements to calculate 1988 Federal income tax. The tax schedule is set up so that you pay 15 percent tax on everything up to a certain income level, which we call the *breakpoint*. Everything above the breakpoint is taxed at 28 percent. The value of the breakpoint, however, depends on your filing status. It's $23,900 if you file as Head of Household, $17,850 if you file as Single, $29,750 if you file as Married, Joint Return, and $14,875 if you file as Married, Separate Return. Obviously, this is fertile ground for using `if then` statements!

Example 26:

1. Go to the If Then card of your HyperTalk Bible stack and create a new button called `Taxes`.
2. Open the button script and insert this text:

```
on mouseUp -- Taxes button
  ask "What is your annual taxable income?"
  if it is empty then exit mouseUp
  else if it < 0 then
    answer "The IRS has a $500 fine for frivolous filings."
    exit mouseUp
  else
    put it into income -- save user input
  end if
  answer "Which schedule?" with "Head" or "Single" or "Married"
  if it is "Head" then
    put 23900 into breakpoint
    put 3585 into baseamount
  else if it is "Single" then
    put 17850 into breakpoint
    put 2677.50 into baseamount
  else
    answer "Separate or Joint Return?" with "Separate" or "Joint"
    if it is "Separate" then
      put 14875 into breakpoint
      put 2231.25 into baseamount
    else
      put 29750 into breakpoint
      put 4462.50 into baseamount
    end if
  end if
  if income <= breakpoint then
    put income * 0.15 into taxes
  else
    put baseamount + (income - breakpoint) * 0.28 into taxes
  end if
  set the numberFormat to "0.00"
  answer "Your tax liability is $" & taxes
end mouseUp
```

3. Click the Taxes button.

Result:

You see the dialog box shown in Figure 9-21.

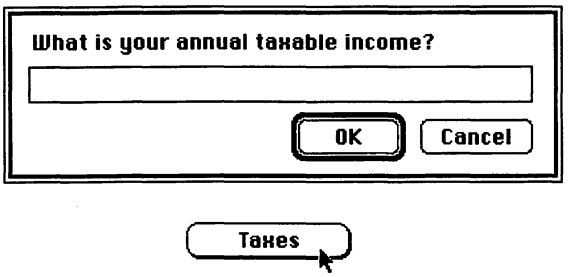


Figure 9-21. Income dialog box.

4. Enter

   30000

   for your annual income.

Result:

Now, you see the dialog box shown in Figure 9-22.

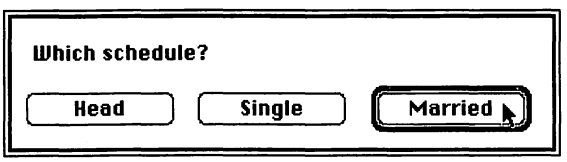


Figure 9-22. Status dialog box.

5. Click the Married box.

Result:

Now you see the dialog box shown in Figure 9-23.

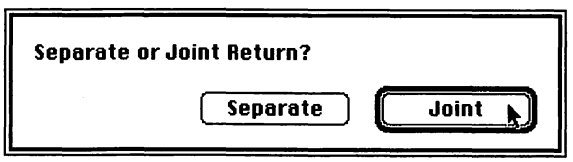


Figure 9-23. Separate/Joint dialog box.

6. Click the Joint box.

Result:

An answer box (see Figure 9-24) reports your tax liability.

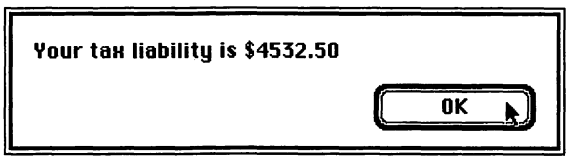


Figure 9-24. Tax liability answer box.

Now, for some comments. First, the `ask` command is the natural choice for obtaining the income. We used some data checking to screen this value, and if valid, we save it in the variable income. (Actually, not having covered strings yet, we didn't check that the input was a number and not something like `None of your business`.)

Next, we need to get the filing status. We could have used `ask`, but it would be difficult to anticipate all the possible answers people might give. By using `answer`, we limit the answers to a known set of responses. Here is a problem, however. There are four filing status categories, but the `answer` command lets you specify only three answers. To get around that, we lumped Married, Joint and Married, Separate into one category: Married. Then, if the user clicks that button, the script puts up a new dialog box to get the rest of the story.

Once we get the data, we proceed to the calculations. We use an `if then else` sequence to set the variable breakpoint to the proper value. Also, we set the variable baseamount to 15 percent of breakpoint. Then, if the income is above breakpoint, the user pays baseamount + 0.28 * (income-breakpoint). That is, the taxed person pays 15 percent of everything up to the breakpoint and 28 percent on everything above the breakpoint.

After setting these values, the script compares income to breakpoint and calculates the tax accordingly.

Finally, to make the output more readable, set the number format to two decimal places. We borrowed the & operator from chapter 11, "Strings." This operator lets you join some words to the value of income to make a single string that the `answer` command can use.

Review Questions

1. Suppose `sam` and `sally` are variables. How do you express the following relationships?

   a. `sam` and `sally` have the same value.
   b. `sam` and `sally` have different values.
   c. `sally` has a greater value than `sam`.
   d. `sam` is greater than `sally` but less than 40.

2. What, if anything, is wrong with each of the following?

   a. `if it <> empty put it into the message box`
   b. `if it is "Yes" then put "You ask too much of me" into the message box`
   c.
   ```
   if the mouseH > 200 then
     beep 3
   else play "boing"
   ```
   d.
   ```
   if the shiftKey is down
     play "boing"
   else
     play "harpsichord"
   ```

3. How would you express the following conditions?

   a. The mouse is located at x = 200, y = 300.
   b. Either or both the Shift key and the option key are down.
   c. The mouse has been clicked and the Shift key is down.
   d. Either the Shift key is down or the option key is down, but not both.

4. Write an `if` statement that makes the Mac beep if the command key is down.
5. Write an `if` statement that puts Austin in the Message box if the response to an `answer` command is Capital, puts Texas in the Message box if the answer response is State, and quits the mouseUp handler if the answer response is Cancel.

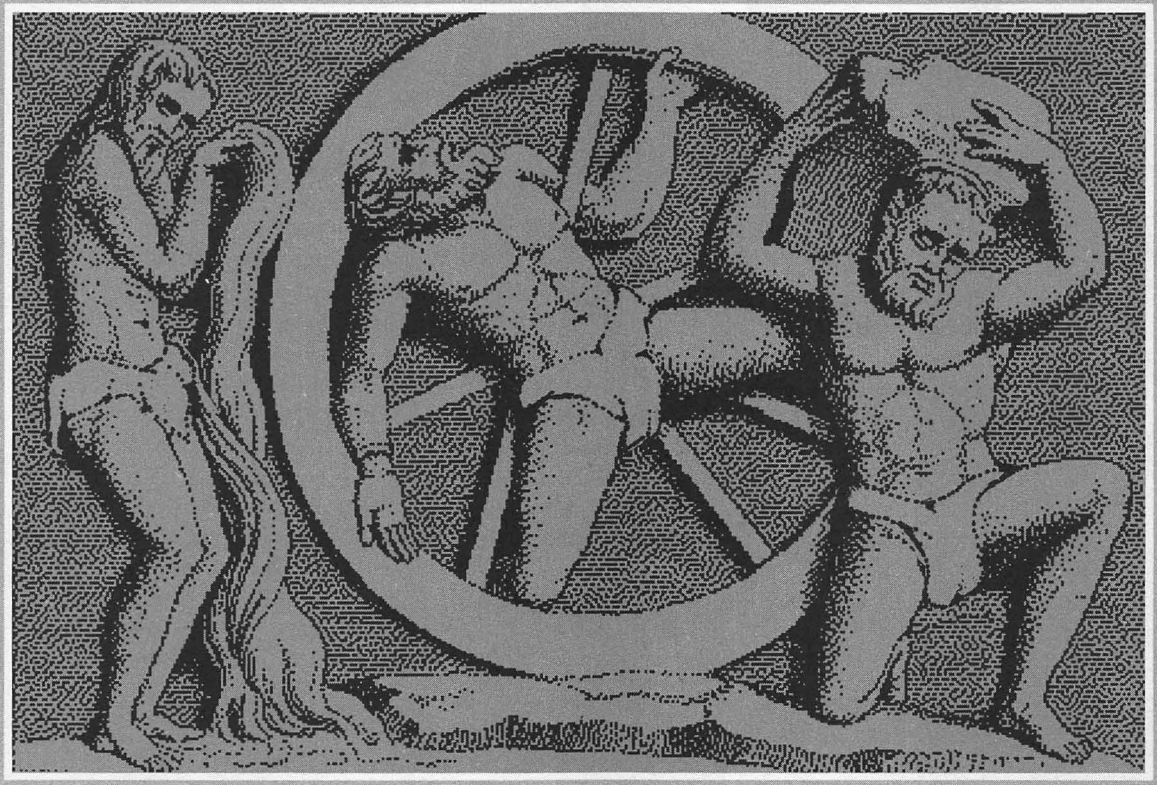
Exercises

1. Write a button script that takes you to the next card in the stack if you hold down the Shift key while you click the button.
2. Write a button script that takes you to the next card in the stack if you hold down the Shift key while you click the button and takes you to the last card in the stack if you hold down the option key while clicking the button.
3. Write a card script that puts up a dialog box asking your name. If you enter a name, the script places it in the first card field. If you select the Cancel button, the script quits.
4. Write a field script that behaves as follows:

   a. If the mouse enters the field and the field is empty, the script puts up an ask dialog box asking `"What's your favorite dessert?"` The script puts `"My favorite dessert is "` into the field, then puts the answer after the field. Thus, if the user replies with Figs, the phrase My favorite dessert is figs appears in the field. If the user makes no entry or selects Cancel, the script puts a default response into the field.
   b. If the mouse enters the field and the field is not empty, the script puts up an ask dialog box asking `"Name another dessert you like:"`. If the user makes no entry or selects Cancel, terminate the script. Otherwise, append `" I also like "` and the user's answer to the field.

Chapter 10

Repeats



Quick View

Commands

`wait`

Keywords

`repeat with`
`repeat for`
`repeat while`
`repeat until`
`repeat forever`
`end repeat`
`exit repeat`
`next repeat`

Concepts

Fixed iteration loops
Repeat control structures
Down-to-type counting
Nested loops
Conditional loops
`repeat while` versus `repeat until`
Messages versus states
Modifying loop behavior
Delaying with `wait`

Chapter 10

Repeats

As you develop your own HyperCard stacks, most likely you will have to repeat a set of commands several times. HyperTalk, like most programming languages, provides tools for repeating a sequence of instructions. We call these repeated instructions a *loop*. Loops are among the most useful and powerful tools for the programmer. They let you automate tasks that otherwise would be hopelessly tedious and time-consuming. Add 3000 numbers? Cycle your Macintosh screen through a set of patterns? No problem when you know how to loop. In this chapter, we explore how HyperTalk implements this essential programming concept. We look at the several forms of the `repeat` command and at ways to terminate a loop. Also, we discuss the `wait` command (no relation to the coauthor). It provides ways to make the computer pause for a while, and it often is used with loops.

Built-In Looping

Some HyperTalk commands have a built-in ability to repeat. For example, `beep 4` makes the Macintosh beep 4 times, and the command `show all cards` gives you a quick glimpse of each card in a stack in turn. Most of the commands, however, do not have the repeat capability built in. That's where the `repeat` command comes in handy, for it lets you repeat a set of commands as many times as necessary. HyperTalk offers several variations on the repeat theme. The simplest repeats a group of commands a predetermined number of times. Other forms of the command continue until some event, such as a mouse click, signals time to stop. Let's look at these variations on a theme now.

Fixed Iteration Loops

Loop structures fall into two general groups: those that repeat (or *iterate*) a fixed number of times, and those that repeat until some true-false condition is met. HyperTalk has two of each type. We look first at those that repeat a given number of times.

The *repeat for* Control Structure

The `repeat for` structure is the simplest form of HyperTalk looping structure, and one you'll use often. Figure 10-1 shows its syntax.

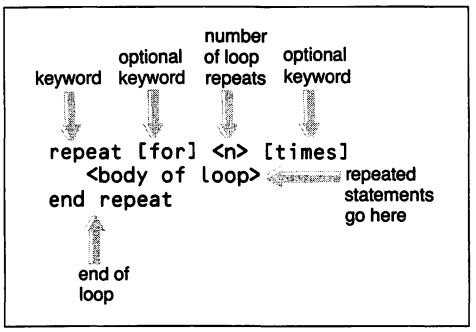


Figure 10-1. The `repeat for` loop syntax.

In the first line of the loop syntax, the `for` and the `times` are optional. Thus, `Repeat 5` works just as well as `repeat for 5 times`, but the optional words make the command absolutely clear. We use the full form.

Using the `repeat for` command in a script is easy. First, type `repeat for` followed by the number of repeats you want and by the keyword `times`. Next, begin a new line and type the commands you wish to repeat, one command to a line. Note that the script editor automatically indents these lines. Finally, type `end repeat` on the line following the final command.

We call the block of commands between the `repeat for` line and the `end repeat` line the *body* of the loop. All the repeat loops follow this general form: `repeat line`, body, `end repeat` line.

Suppose, for instance, you want five beeps to be generated when you click a particular button, but you want them to be spaced farther apart than `beep 5` provides. You can use the `wait` command (see chapter 3, "Audio and Visual Effects") to provide the pause. Here's our first loop example.

Example 1:

1. Go to your HyperTalk Bible stack and create a new card called `Repeats`.
2. Create a new button called `Repeat For`, and put the following script into the button:

```
on mouseUp -- Repeat For button
  repeat for 5 times
    beep 1
    wait for 20 ticks -- wait 20 "ticks" before continuing
  end repeat
end mouseUp
```

3. Use the browse tool to click the button.

Result:

Your Macintosh beeps, pauses briefly, beeps again, and pauses again. This cycle occurs five times.

Pressing the Tab key after the `end repeat` line causes the HyperTalk script editor to align everything. You can also just press the Return key, but then you get an unsightly blank line.

The design of this command is pretty clear, but let's go through its elements. The `repeat for` and the `end repeat` lines act as boundaries for this structure. Everything between these two lines is repeated as a unit. That is, HyperTalk executes the `beep` command, then the `wait` command, then starts the cycle (or loop) over again. It does not do five beeps then five waits. Figure 10-2 shows a flow chart for the command.

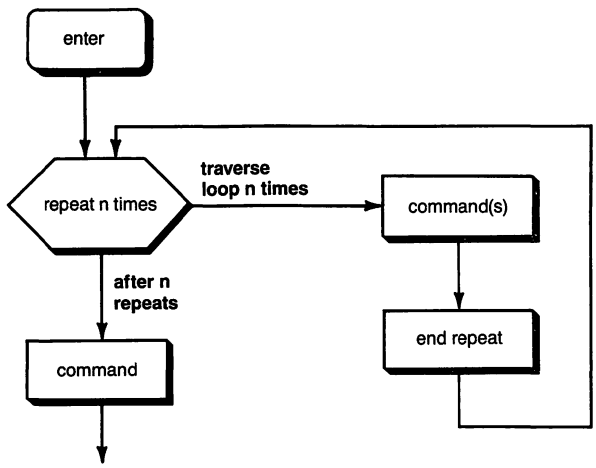


Figure 10-2. Flow chart for `repeat for`.

The number of times the loop repeats (`<n>` in the syntax) can be provided by any container or expression that reduces to an integer. For instance, suppose that you want a button that shows each card in a stack and then stops. If the stack had 10 cards, you could use `repeat 10`, but then you would have to change the script each time you changed the size of the stack. A much better approach is to use the `number of` function to get the number of cards in the stack. The next example shows how.

Example 2:

1. Go to the Repeats card of your HyperTalk Bible stack and create a new button called `Show Stack`.
2. Place the following handler into the button script.

```
on mouseUp -- Show Stock button
  repeat for the number of cards
    visual effect dissolve to grey
    beep
```

continued

```
    go to next card
  end repeat
end mouseUp
```

3. Click the button.

Result:

Each card in the stack is displayed in turn, punctuated by a dissolve effect and a beep. The program ends when the original card is shown.

Again, each command in the body of the loop is executed once and then the loop repeats. In this case, the number of repeats equals the number of cards in the stack, so the last `go to next` card brings you back to the original card.

The *repeat with* Control Structure

Suppose that you need a slightly more complicated loop, such as one that beeps once the first cycle, twice the second cycle, and so on. The `repeat with` command makes that a simple matter. It creates a new variable that increases by one each cycle of the loop. Figure 10-3 shows the syntax.

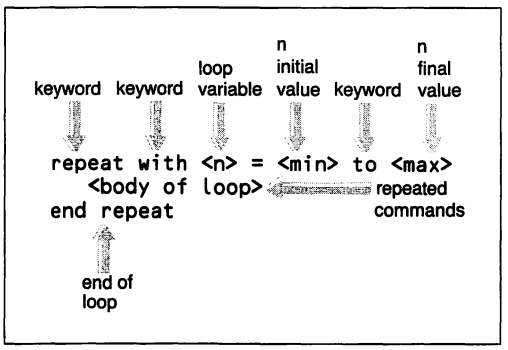


Figure 10-3. The `repeat with` loop syntax.

Here `n` is a HyperTalk variable created by the loop and used for counting; you're free to make your own choice for the name of this variable. The variable `n` starts off with the value `min`, and HyperTalk automatically adds 1 to `n` during each cycle of the loop until `n` reaches `max`. Let's look at an example.

Example 3:

1. Go to the Repeats card of your HyperTalk Bible stack and create a new button called `Repeat With`.
2. Put this script into the button:

```
on mouseUp -- Repeat With button
  repeat with beepcount= 1 to 5
```

continued

```
    beep beepcount
    wait 20
  end repeat
end mouseUp
```

3. Click the mouse in the button.

Result:

Your Macintosh will beep once, pause, beep twice, and so on, up to five beeps.

Here we chose the name `beepcount` for the variable. In the loop, `beepcount` takes the values `1` through `5` in turn.

The loop's limits `min` and `max` can be integer constants, as in our example, or any containers or expressions that reduce to constants. For instance, you can use limits like `20 to 50`, `-10 to 10`, `1 to the number of cards`, and so on. Figure 10-4 shows a flow chart for this command.

Let's look at another example, one in which the loop limit is given by a variable. This example uses the `play` command to play a number of notes equal to the user's response to an ask dialog box.

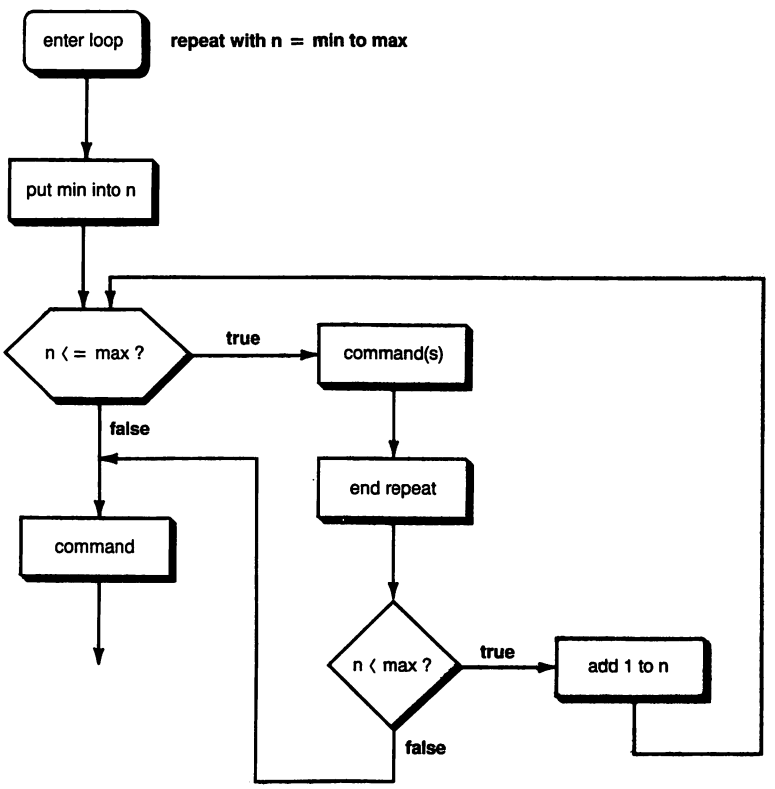


Figure 10-4. Flow chart for `repeat with`.

Example 4:

1. Go to the Repeats card of your HyperTalk Bible stack and create a new button called `Repeat Notes`.
2. Place the following script into the button:

```
on mouseUp -- Repeat Notes button
  ask "How many notes?" -- reponse put into global variable it
  repeat with notevalue = 60 to 60 + it - 1
    play "harpsichord" tempo 300 notevalue
  end repeat
end mouseUp
```

3. Click the button and enter 6 in the ensuing dialog box.

Result:

The Macintosh plays 6 notes ascending in pitch.

Here, note number 60 is middle C. If you enter `1`, the expression `60 + it - 1` reduces to the value `60`. The loop limits become `60 to 60`, and the loop is executed just once, with `notevalue` set to 60. If the input is `2`, the limits become `60 to 61`, and the loop is executed twice, using note number 60 the first time and note number 61 the second time. The variable `notevalue` controls both the number of loops and the pitch of the notes being played. We used a tempo of 300 to make the script run faster.

Counting Backward

A loop that begins with `repeat with n = 60 to 60`, as you have seen, loops just once. What happens if you use `repeat with n = 60 to 50`? Try Example 4, and use a value of `-9`. That value for the global variable `it` makes the loop limits `60 to 50`. Does the loop count backwards? No, for if the second limit is less than the first, the loop is skipped entirely. What if you want to process a field from the last line to the first or go from high notes to low? To make a loop variable decrease rather than increase, use the `down to` variation of the `repeat with` command. Figure 10-5 shows its syntax:

Look at a simple example.

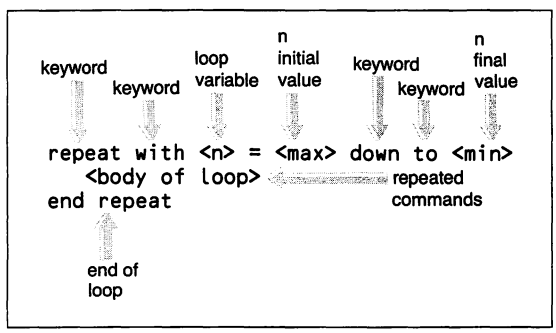


Figure 10-5. The `repeat with ... down to` structure syntax.

Example 5:

1. Go to the Repeats card of your HyperTalk Bible stack and make a new button called `Count Down`.
2. Put the following script into the button:

```
on mouseUp -- Count Down button
  repeat with beepcount = 5 down to 1
      beep beepcount
  wait 1 seconds
  end repeat
  play "boing"
end mouseUp
```

3. Click the mouse in the Count Down button.

Result:

Your Macintosh beeps 5 times, waits a second, beeps 4 times, and so on. After the final beep, the Macintosh waits a second and boings.

Here, we've used the keyword `seconds` to make `wait` count in seconds instead of in ticks. Note that we have to use the plural, even if the number is 1.

Figure 10-6 shows the flow diagram for this loop.

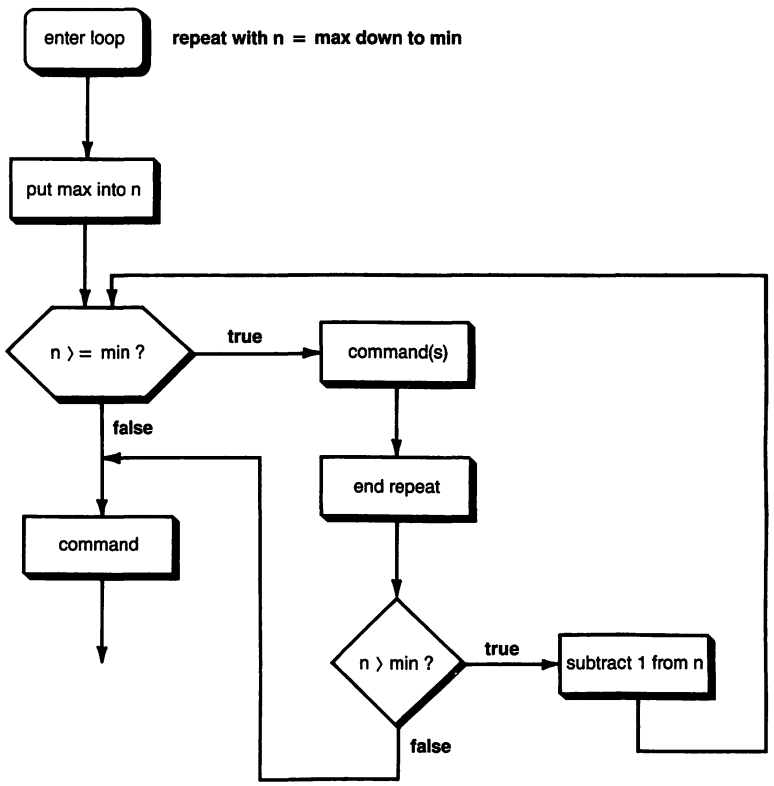


Figure 10-6. Flow diagram for `repeat with ... down to`.

As with the other loops, the limits for a `down to` loop can be integers or containers or expressions that reduce to integers. The only additional requirement is that the first limit be equal to or greater than the second limit; otherwise the loop is skipped.

We can create an interesting example with fields that process the lines in a field.

Example 6:

1. Go to the Repeats card of your HyperTalk Bible stack and create a new button called `Reverse Fields`.
2. Enter the following handler in the button script:

```
on mouseUp Reverse Fields button
  put the number of lines in card field "Source" into maxlines
  repeat with lineno = maxlines down to 1
    put line lineno of card field "Source" into line maxlines + 1 ¬
    - lineno of card field "Target"
  end repeat
end mouseUp
```

3. Create two card fields, one called `Source` and one called `Target`. Enter two or more lines of text in Source. Press the Return key to start a new line. Resize the two new fields, if necessary, so each is large enough to hold the text.
4. Click the Reverse Fields button.

Result:

The text in card field Source is copied to card field Target, with the last line of Source going into the first line of Target, and so on. See Figure 10-7 for an example.

This script is more involved than the previous example, so look at some of the details. First, the script uses the `number` function to find out how many lines of text there are in field Source; this value is saved in the `maxlines` variable. The `repeat` loop then counts down, starting with the last line in field Source. To see how the body of the loop works, suppose there are three lines. Then the loop variable `lineno` starts off with the value `3`. Also, the expression `maxlines - lineno + 1` has the value `3 - 3 + 1`, or `1`, so the first pass through the loop copies line 3 of field Source into line 1 of field Target. Each additional cycle decreases the line number for field Source and increases it for field Target, so the lines are copied in reverse order.

By the way, if you just type text into a field without pressing the Return key, the text "wraps" to the next line on the screen, but it still belongs to the same line logically. You need to use the Return key to produce true lines.

Nested Loops

What kind of commands can you have in a loop? Any kind, including other loops! When one loop is inside another, we say it is "nested." With nested loops, the inside loop runs through its full gamut of cycles for each one cycle of the outer loop.

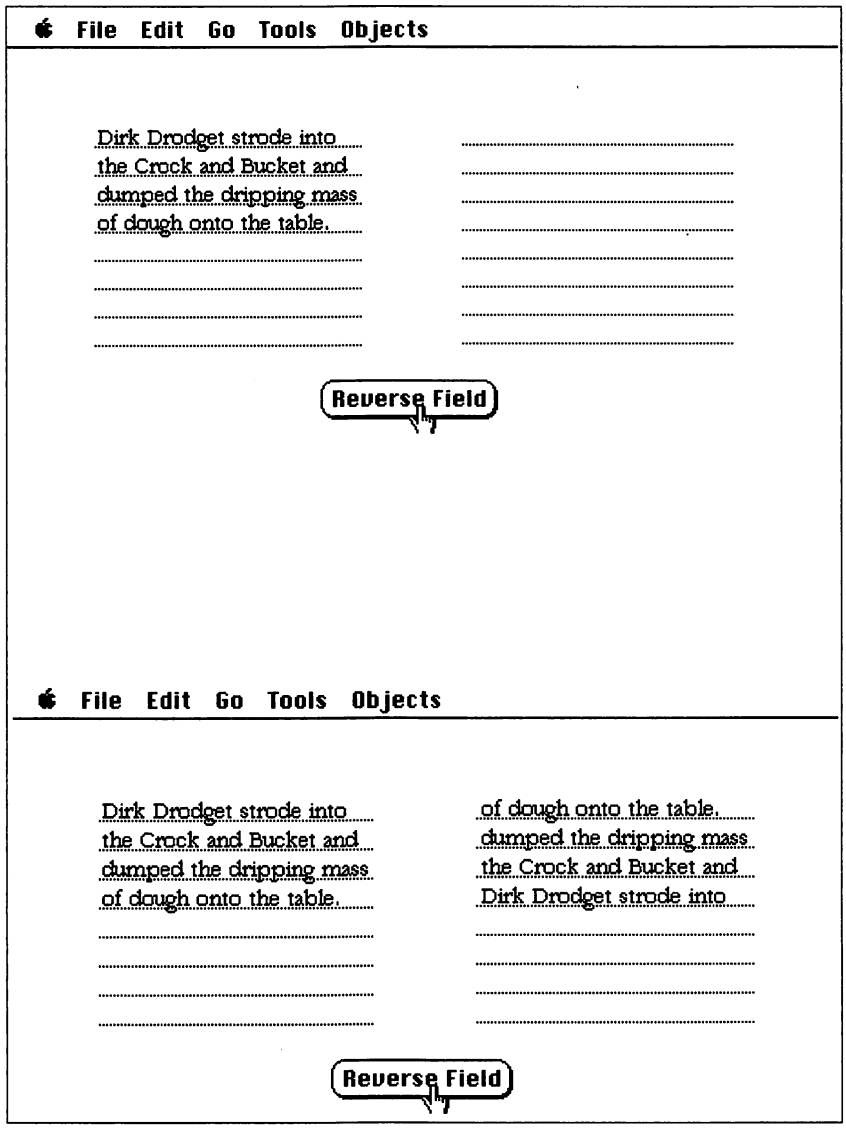


Figure 10-7. Reverse fields example.

Nested loops are handy for tasks such as processing a bunch of fields in a bunch of cards. One loop handles all the fields in a given card, and the other loop cycles through the cards. Let's study a simple example to see how nested loops work.

Example 7:

1. Go to the Repeats card of your HyperTalk Bible stack and create a new button called `Nested Loop`.
2. Put the following script into the button:

```
on mouseUp -- Nested Loop button
  repeat for 5 times                    -- outer loop starts
    repeat with note = 84 to 86         -- inner loop starts
      play "harpsichord" tempo 400 note
    end repeat                          -- inner loop ends
    play "boing" tempo 200 63
  end repeat                            -- outer loop ends
end mouseUp
```

3. Click the mouse in the Nested Loop button.

Result:

Your Macintosh plays five groups of a three-note harpsichord pattern and a boing.

The `repeat with note = 84 to 86` loop the "inner" loop, and the `repeat for 5 times` loop is the "outer" loop. You can hear what each loop does. For each single cycle of the outer loop, the inner loop cycles three times, producing the three-note harpsichord pattern. The boing in the outer loop punctuates the finish of each complete cycling of the inner loop. Figure 10-8 illustrates this nested loop behavior.

```
on mouseUp
  repeat for 5 times
     repeat with note = 84 to 86
         play "harpsichord" tempo 400 note
    end repeat
     play "boing" tempo 200 63
  end repeat
end mouseUp
```

Figure 10-8. Nested loops.

By using two `repeat with` loops rather than one `repeat for` and one `repeat with`, we can produce a more complex behavior. The next example, for instance, uses one loop to control the tempo of a `play` command and a second loop to control which notes are played:

Example 8:

1. Go to the Repeats card of your HyperTalk Bible stack and create a button called `Play Notes`.
2. Put the following script into the button:

```
on mouseUp -- Play Notes button
  repeat with speed = 1 to 10 -- sets tempo
    repeat with note = 66 down to 54 -- sets notes
      play "harpsichord" tempo 100 + 50 * speed note
    end repeat
  end repeat
end mouseUp
```

3. Click the mouse in the Play Notes button.

Result:

Your Macintosh plays a series of descending notes 10 times, each time with increasing speed.

Here the note value is given by `note`, the loop variable for the inner loop. In this handler, the tempo is set by the expression `100 + 50 * speed`, where `speed` is the loop variable for the outer loop. While the inner loop cycles once through its note values, speed stays constant. But after the inner loop finishes, speed is updated in the outer loop to the next value, setting the stage for the next run of notes. You could use the same script in an on openCard handler to add pizazz to opening a particular card in a stack.

Sending a Message from the Message Box

Anytime you want to send a message to a handler, you can do so manually from the Message box. All you do is type the name of the handler into the Message box. For example, suppose that your openStack handler initializes several variables, and you have reworked the script and wish to initialize them again. You don't have to go to another stack and then come back to get the openStack to fire up. Just type `openStack` into the Message box. Your stack starts as though this were the first time it was run. All global variables are still intact with their last values.

Practical Nested Loops

Okay, so nested loops can help you make interesting sound effects. Can they help with more mundane projects? Of course. For instance, suppose that you want to find the total number of words in all the card fields in a stack. The function `the number of words in card field 1` tells us the number of words in a single card field. To add up the number of words we can use a loop like this:

```
repeat with fieldno = 1 to the number of fields
  add the number of words in card field fieldno to wordtotal
end repeat
```

Then, to get the total number of words for all cards, we can nest this loop inside a loop that cycles through the cards:

Example 9:

1. Go to the Repeats card of your HyperTalk Bible stack and create a new button called `Count Words`.
2. Place the following script into the button:

```
on mouseUp -- Couunt Words button
  put 0 into wordtotal -- start off the count at 0
  repeat for the number of cards times
```

continued

```
      repeat with fieldno = 1 to the number of card fields
        add the number of words in card field fieldno to wordtotal
      end repeat
      answer wordtotal
      go to next card
    end repeat
    put wordtotal into the message box
  end mouseUp
```

3. Click the Count Words button.

Result:

HyperTalk counts up all the words in all the card fields and reports the result in the Message box.

This illustrates a typical use for nested loops: processing objects within a list of objects higher in the heirarchy. This particular example just does card fields. By adding a second nested loop, you can also process background fields.

Conditional Loops

The `repeat for` and the `repeat with` commands work best when you know in advance how many times you want a loop to repeat. For ten repeats, just use `repeat for 10 times` or `repeat with n = 1 to 10`. Often, however, you need a loop that continues until some event happens, such as someone clicking a mouse, typing a certain key, or entering a certain response in a field or dialog box. In this case, you can use one of HyperTalk's *conditional loops*. A conditional loop is one that tests a condition to see whether it is true or false before starting each iteration of the loop. If the test fails, the loop terminates; otherwise it continues for one more cycle and tests the condition again. HyperTalk has two conditional loops. The `repeat until` loop continues until a specified condition becomes true; the `repeat while` loop continues as long as a given condition remains true. These loops use the same sort of Boolean tests that the `if` statements do. Let's look at these loops.

The *repeat until* Control Structure

First, look at the syntax for the `repeat until` loop in Figure 10-9.

Like the other forms of `repeat`, the initial `repeat` line is followed by the commands to be repeated and an `end repeat` to mark the end. The Boolean expression is the same sort of expression we discussed in chapter 9.

To see how this form of loop works, construct a button script that cycles through the cards in a stack until you click the mouse. To make this a little jazzier than the `show all` cards command, we add a visual effect.

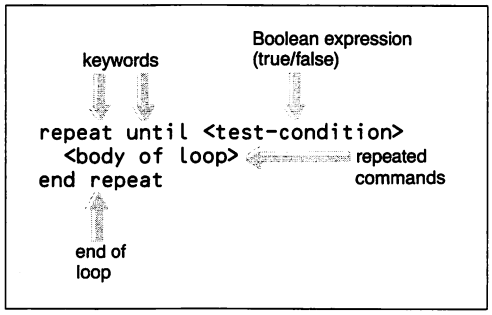


Figure 10-9. The `repeat until` loop syntax.

Example 10:

1. Go to the Repeats card of your HyperTalk Bible stack and select the background mode.
2. Create a background button called `Scan Cards`.
3. Enter the following script in the button:

```
on mouseUp -- Scan Cards button
  repeat until the mouseClick
    visual effect dissolve fast to grey
    go to next card
  end repeat
end mouseUp
```

4. Click the Scan Cards button.

Result:

HyperCard begins dissolving from one card in the stack to the next until you click the mouse. Then it stops changing cards.

This example works best in a stack with several cards of differing appearance. By now, your HyperTalk Bible stack should fit the bill. We put the button in the background so that you can access it from more than one card.

HyperTalk is a clear language. Read the handler, and it's easy to see what it does. The `mouseUp` message causes HyperTalk to go from one card to the next in the stack, doing a dissolve each time. This continues until you click the mouse. One extremely important point is that the loop terminates between cycles, when the true-false condition is tested. If you click the mouse halfway through a loop, the loop keeps going until the test is made. So even if you click during the `visual` command but before the `go to` command, HyperTalk still executes the `go to` command. Then HyperCard tests to see whether the mouse has been clicked. Because it was, the loop ends at that point. See Figure 10-10. If it is important to you to quit in the middle of a loop, you may wish to use the `exit repeat` and `next repeat` commands; we discuss them later in this chapter.

Another key point for this type of loop is that you follow the `repeat until` with an expression that HyperCard can evaluate as being true or false. In this case, we use the `mouseClick` function, which, recall, returns a value of `true` when the mouse

is clicked. More generally, you can use any of the forms of conditional expressions that we discussed in chapter 9, "Ifs." Let's look at a few examples, now.

```
on mouseUp
  repeat until the mouseClick
     visual effect dissolve fast
    go to next card
  end repeat
end mouseUp
```

1. mouse clicked here
2. loop keeps going
3. until the text condition is checked

Figure 10-10. Terminating a `repeat until` loop.

Using a Relational Expression as the Test Condition

The test condition can be a relational expression comparing two values. The next example uses the `mouseH` function to see whether the horizontal position of the mouse is greater than 410; that is, it checks to see whether the mouse is in the right 20 percent (approximately) of the card.

Example 11:

1. Go to the Repeats card of your HyperTalk Bible stack and create a new card called `Mouse Rights`.
2. Open the card script and enter the following handler:

```
on openCard
  repeat until the mouseH > 410
    put "move the mouse to the right" into the message box
    play "boing" tempo 400 80
    wait 1 seconds
    put empty into the message box
    wait 20 ticks
  end repeat
  put "thank you" into the message box
end openCard
```

3. Go to another card, position your mouse near screen center, and return to the Mouse Rights card.

Result:

The Message box appears (if not already there). The message `move the mouse to the right` flashes on and off, and the Macintosh boings until you move the mouse far enough to the right for the boinging to stop.

The script makes the message flash by alternating the message in the box with erasing the message by putting `empty` into the box. The `wait` command ensures that you have enough time to read the message between erasures. Only when the test condition (`the mouseH > 410`) becomes true does the loop halt. Then, the final message is displayed.

Just as with `if then` statements, so also you can use logical operators to com-

bine tests. For instance, if you want the mouse to be in the upper-right portion of the card, you could replace `repeat until the mouseH > 410` with `repeat until the mouseH > 410 and the mouseV < 68`. Of course, you should alter the message in the Message box so that the user knows where to put his or her mouse.

The various key-state functions provide additional ways to control a `repeat until` loop.

Data Checking

The `repeat until` loop can check user input for validity. For instance, suppose that you want input to be limited to numbers from 1 to 10. You can use an `if then` statement to check whether the input meets these conditions, but if you use a `repeat until` loop, you can prompt users for more input until they get it right. Here is a simple example. It sums the integers from 1 to the input integer, but its chief interest for you is in how the script handles the input:

Example 12:

1. Go to the Repeats card of your HyperTalk Bible stack and create a new button called `Sums`.
2. Give it this script:

```
on mouseUp -- Sums button
  ask "Pick a number from 1 to 10"
  if it is empty then exit mouseUp
  repeat until it >= 1 and it <= 10
    ask "PLEASE pick a number from 1 to 10" with "5"
    if it is empty then exit mouseUp
  end repeat
  put 0 into sum
  repeat with n = 1 to it
    add n to sum
  end repeat
  answer sum
end mouseUp
```

3. Click the Sums button; when the dialog box asks you to `Pick a number from 1 to 10`, respond with `20`.

Result:

A new dialog box asks you to `Please pick a number from 1 to 10` and offers a default value of `5`.

4. Respond with `10`.

Result:

An answer box displays `55`, the sum of the integers 1 through 10.

Note the structure for this script. First, before the loop begins, `ask` obtains the user input, which is stored in the global variable `it`. If the answer satisfies the test critereon, the body of the loop is skipped, and the script goes directly to the next phase, calculation. Only when the initial answer is invalid is the `repeat until` loop executed. Here, it is important that the body of loop contain code to update the global

variable `it`. If `it` remains unchanged, the loop condition remains `false`, and the program will get caught in an endless loop. Fortunately, this example has an `ask` command in the loop, and this command obtains a new value for the global variable `it`.

The *repeat while* Control Structure

Now, let's look at the other conditional loop, the `repeat while` loop. Instead of waiting for a condition to become true, the way the `repeat until` loop does, the `repeat while` loop waits for a condition to become false. The `repeat while` loop has the syntax shown in Figure 10-11.

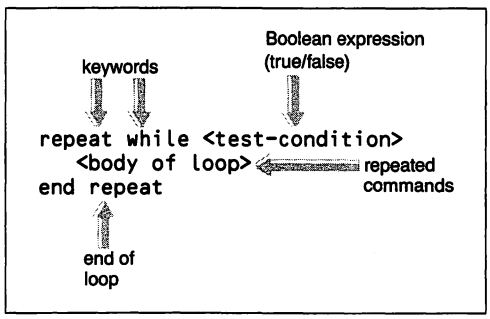


Figure 10-11. The `repeat while` loop syntax.

The `repeat while` loop continues as long as `<test condition>` is true and quits when the condition becomes false. To see how that works, let's build an icon-moving script.

Example 13:

1. Go to the Repeats card of your HyperTalk Bible stack and create an `Eye Move` button.
2. Give the button the following script:

```
on mouseDown -- mouseDown handler instead of mouseUp
  get location of me
  put it into start
  repeat while the mouse is down -- while instead of until
    add 10 to item 1 of it
    set loc of me to item 1 of it
    get loc of me
  end repeat
  set loc of me to start
end mouseDown

on mouseUp
  -- trap the mouseUp message
end mouseUp
```

3. Click the Eye Move button, but take your time in releasing the mouse button.

Result:

The eye icon moves to the right until you release the mouse button, and then it returns to its starting point.

How does it work? First, notice that the loop is contained in a mouseDown handler. Thus, the handler goes into action as soon as you press the mouse button. Next, look at the loop structure. Here the test condition is `the mouse is down`. Unless you have a very quick finger, the mouse is still down when the script reaches this line. That makes the condition true, so the loop starts cycling, moving the button across the screen. Once you release the mouse button, the test becomes false, the loop terminates, and the script restores the button to its original position.

The mouseUp handler at the end of the script is there to catch the `mouseUp` message and prevent it from being passed along the hierarchy. Why? You could have set up your stack to process the `mouseUp` message when the mouse is clicked outside a button. For example, your background script could include a handler like this:

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

Then, if you click the card outside a button, you're taken to the next card. If you click in a button, the handler intercepts the `mouseUp` message first, and you stay on the same card.

repeat until Versus *repeat while*

The `repeat while` loop is very similar to the `repeat until` loop. See Figure 10-12. Each is what programmers term an *entry condition loop.* That is, the loop structure evaluates the test condition before each cycle of the loop is started. This "look before you loop" feature means that if the test condition is false to begin with, the whole loop structure is skipped.

Logically, you only need one of these two forms, because one can be converted to the other by changing the test condition. For instance, `repeat until short name of this card is short name of last card` is equivalent to `repeat while short name of this card <> short name of last card`.

More generally,

```
repeat while test-condition
```

is the same as

```
repeat until not-test condition
```

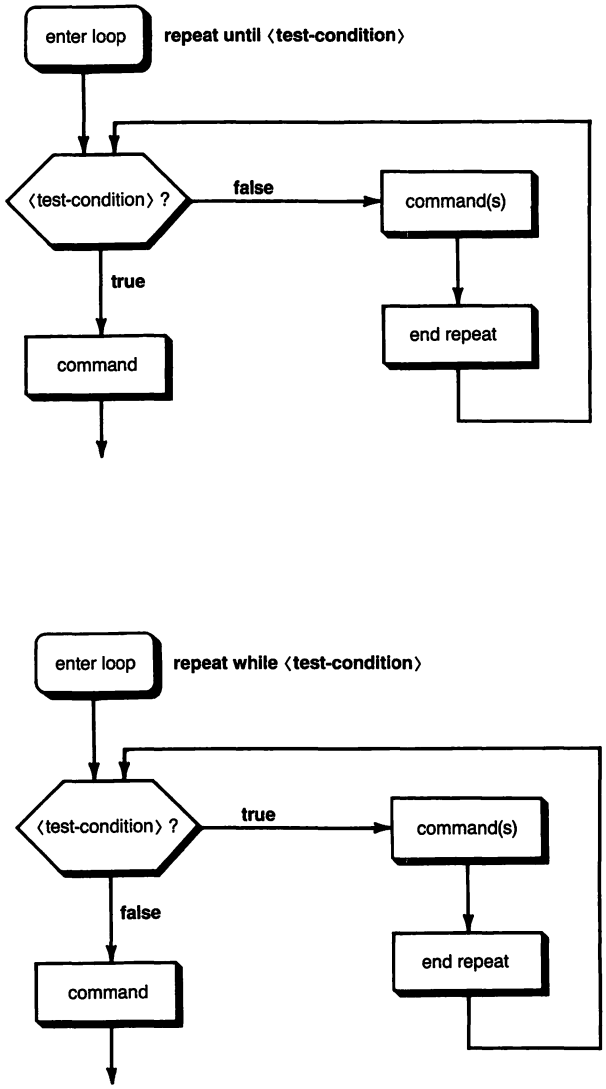


Figure 10-12. Flow charts for `repeat until` and `repeat while`.

Thus, `repeat until the shiftKey is down` is the same as `repeat while the shiftKey is up`, and `repeat until the mouseH > 410` is the same as `repeat while the mouseH <= 410`. Or you can use the `not` operator and have `repeat while not (the mouse > 410)`. Which loop form you use is a matter of deciding which reads the most clearly.

Messages and States

One distinction you should be sure to understand is the one between `mouseDown` and `the mouse is down`. `mouseDown` is a message, an event, a happening. Pressing the mouse button sends the `mouseDown` message through the object heirarchy, and the message activates the first on mouseDown message handler it encounters. `the mouse is down`, on the other hand, is a Boolean expression that tests the state of the mouse button. The message happens at a particular moment, whereas the state persists until another action changes it. When you press the mouse button, the `mouseDown` message is sent, *and* the mouse state is set to down. The state then remains down until a `mouseUp` message comes along. Let's look at a specific example. Here is an excerpt from Example 13:

```
on mouseDown
  get location of me
  put it into start -- save original position
  repeat while the mouse is down
    :
    :
end mouseDown
```

Suppose that you press the mouse button down with the mouse cursor over the button holding this script. That sends the message that starts the script. When the script gets to the `repeat` statement, it doesn't look for another `mouseDown` message. Instead, it uses the `mouse` function to check whether the mouse state still is down.

What happens if you press the mouse button, and then release it and press it so rapidly that the mouse state changes from down to up and back to down before the program reaches the `repeat` statement? Does the `repeat while` loop start? Yes, it does, because `the mouse is down` checks the mouse condition at the moment the test is made.

What if you use `the mouseClick` rather than `the mouse is down`? In that case, your timing is not as important. You can think of `the mouseClick` function as asking "Has there been a mouseclick?" If the mouse is clicked quickly during the loop, therefore, `the mouseClick` function remembers that and returns true later when the test is made.

One other point to keep in mind is that a `mouseDown` message initially goes to the button or object immediately under the mouse cursor. The `mouse` function, however, is a global function. It returns the current mouse condition regardless of the location of the mouse cursor.

Modifying Loop Behavior

The `repeat for` and `repeat with` loops iterate a predetermined number of times, whereas the `repeat until` and `repeat while` loops iterate a number of times deter-

mined by a test condition that's checked before each loop. Sometimes you may want to skip part of a loop or exit a loop prematurely because of something that has occurred during the loop. Then you can modify the basic behavior of the loops by using the `exit repeat` and the `next repeat` statements. Let's see how they work.

The *exit repeat* Command

The `exit repeat` command causes a loop to terminate immediately. Often it is used to provide a second way of terminating a loop. Figure 10-13 illustrates how this form works.

```
on mouseUp
  repeat for 1000 times
    beep 2
     if the shiftKey is down then exit repeat
     play "boing"
  end repeat
  play "harpsichord"
end mouseUp
```

Figure 10-13. The `exit repeat` command.

Let's look at a simple example; it has a loop that terminates when the loop variable reaches 10 or when you click the mouse.

Example 14:

1. Go to the Repeats card of your HyperTalk Bible stack and create a button called `Exit Repeat`.
2. Enter the following handler in the button script:

```
on mouseUp -- Exit Repeat button
  repeat with n = 1 to 10
    beep n
    wait 20
    if the mouseclick then exit repeat -- alternative exit
  end repeat
  play "boing"
end mouseUp
```

3. Click Exit Repeat.

Result:

If you do nothing else, the loop runs ten cycles, making one more beep each cycle. If you click the mouse, the loop terminates at that point.

You can use `exit repeat` to construct loops that provide a choice of actions, depending on how the loop is exited. Consider the following script, for instance:

```
on mouseUp
  repeat until the mouseclick
    visual effect dissolve fast
    go to next card
    if the shiftKey is down then
      go to first card
      exit repeat
    else if the optionKey is down then
      go to last card
      exit repeat
    end if
  end repeat
end mouseUp
```

It shows all the cards in the stack repeatedly. If you click the mouse, the display stops on the current card. If you depress the Shift key, the loop terminates, and HyperCard goes to the first card of the stack. Or you can press the option key to terminate the loop and go to the last card of the stack.

FOR-NEXT-Like Loop
HyperTalk contains statements that work much like the FOR NEXT statements used in BASIC for looping. HyperTalks `end repeat` works in a repeat loop like the NEXT statement in the FOR loop. The `end repeat` sends execution to the top of the repeat loop. The `end repeat` works like BASIC'S GOTOnnn, where nnn is equivalent to the label of the next statement in HyperTalk's repeat loop.

The *next repeat* Command

The `next repeat` command resembles `exit repeat`, but it is different in one important detail. Instead of making the loop terminate, `next repeat` causes a loop to skip over the remaining commands in the loop and to start the next loop cycle. Figure 10-14 illustrates this point.

```
on mouseUp
  repeat for 1000 times
    beep 2
     if the shiftKey is down then next repeat
     play "boing"
  end repeat
  play "harpsichord"
end mouseUp
```

Figure 10-14. The `next repeat` command.

Here's a simple example that gives you audio feedback on how the command works:

Example 15:

1. Go to the Repeats card of your HyperTalk Bible stack and create a button called `More Beeps`.
2. Enter the following handler in the button script:

```
on mouseUp -- More Beeps button
  repeat until the mouseClick
    repeat for 2 times
      beep
      wait 20 ticks
    end repeat
    wait 40 ticks
    if the shiftKey is down then next repeat -- here it is
    repeat for 4 times
      beep
      wait 20 ticks
    end repeat
    wait 40 ticks
  end repeat
end mouseUp
```

3. Click the More Beeps button.

Result:

Your Macintosh emits a two-beep burst, a four-beep burst, then a two-beep burst, and so on.

4. Click the mouse.

Result:

The beeping stops.

5. Click More Beeps again; this time hold the Shift key down for a while.

Result:

This time the Macintosh only emits the two-beep bursts. Only when you release the Shift key do you get the two-beep, four-beep pattern.

The important point to notice is that `next repeat` does not terminate the loop; it simply causes part of the loop to be skipped.

Now that you see how it works, let's look at a more interesting example. The next script will take you to each card in a stack and give you the option of deleting it. It uses the `answer` command to show a dialog box asking if you want to cancel (i.e., quit the script), delete the card, or leave the card. Then the `next repeat` and `exit repeat` commands implement the response.

Example 16:

1. Go to the Repeats card of your HyperTalk Bible stack and create a few extra cards so that you'll have some available to delete.
2. Create a new button called `Remove Cards`.
3. Enter the following handler in the button script:

```
on mouseUp -- Remove Cards button
  repeat with n = 1 to the number of cards
    go to card n
    answer "Delete this card?" with "Cancel" or "Yes"¬
    or "No"
    if it is "Cancel" then
      exit repeat
    else if it is "No" then
      next repeat
    end if
    play "boing"
    doMenu "Delete Card"
  end repeat
end mouseUp
```

4. Click the Remove Cards button.

Result:

HyperTalk takes you to the first card of the stack. You are asked whether you want to delete the card. If you click the No button, the loop cycles to the next card and repeats the question. If you click the Yes button, the script deletes the current card, boings, then goes to the next card. If you click the Cancel button, the loop terminates. Figure 10-15 shows the flow chart for this script.

The important points are how `exit repeat` and `next repeat` commands are used. The script checks to see whether the user's response (stored in the global variable `it`) is Cancel. If so, the `exit repeat` command terminates the loop. If not, the `if then else` structure next checks to see whether the response is No. If so, the script uses `next repeat` to skip the rest of the loop. If the user's response is neither Cancel or No, it must be Yes. In that case, the final lines of the loop are not skipped, and the script uses the `doMenu` command to delete the card. The Macintosh also boings to let you know that something has happened.

Exiting the Whole Script

The `next repeat` command skips over part of a loop and starts the next cycle. The `exit repeat` command terminates the loop altogether, moving script flow to the next command following the loop. What if you want to quit the whole script? Then, you can use the `exit <message-name>` command we introduced in chapter 9, "Ifs." See the Summary Script at the end of this chapter for an example.

repeat or repeat forever

A loop starting with a lone `repeat` or a `repeat forever` continues until an `exit repeat` or an `exit HyperCard` is encountered. Thus, it is easy to build loops that exit when a certain complex decision is satisfied.

```
repeat forever
   put y * 5 + trun( bigZ ) into little Z
   if littleZ = bigZ then
      if bigZ < oldZ then exit repeat
   end if
end repeat
```

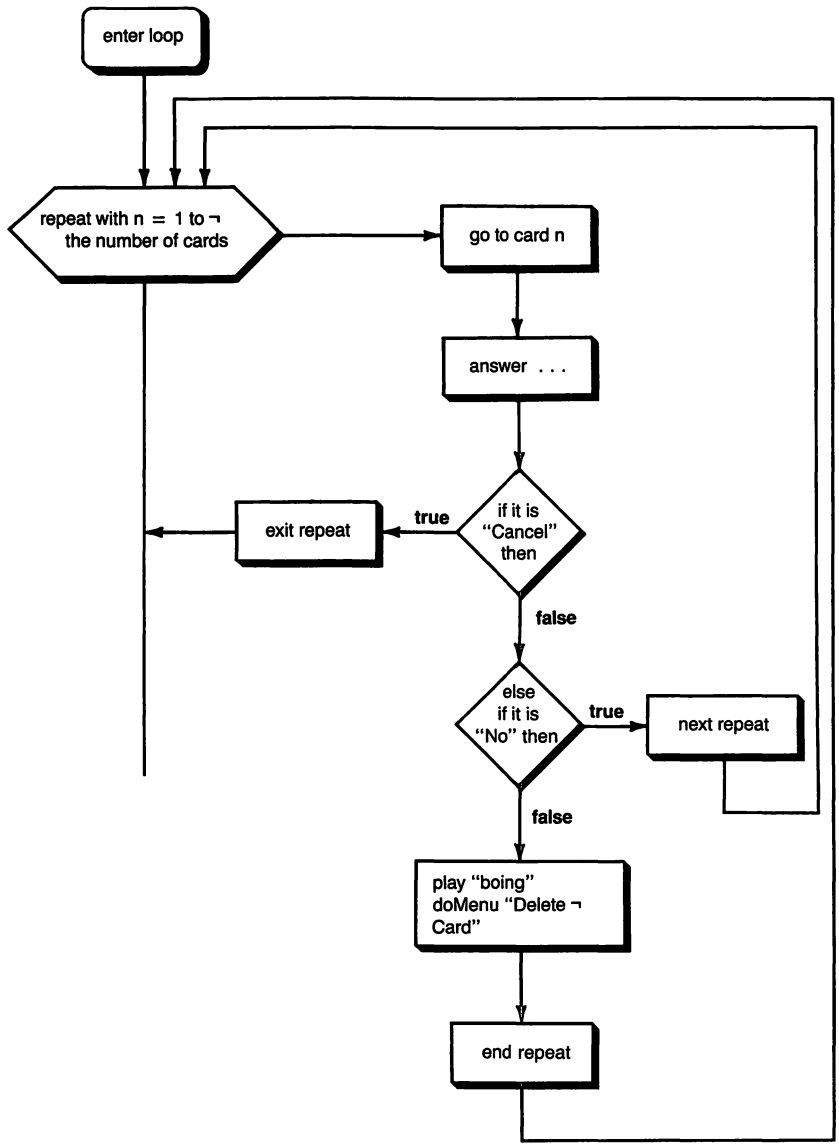


Figure 10-15. The Remove Cards script.

The *repeat forever* Command

HyperTalk, at least Versions 1.01 through 1.2, has an undocumented loop called `repeat forever`, the syntax of which is shown in Figure 10-16.

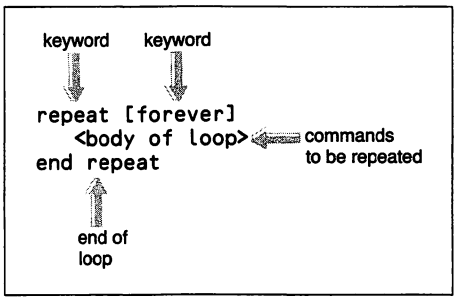


Figure 10-16. The `repeat forever` command syntax.

As its name implies, it keeps going. Note that the `forever` is optional, so a simple `repeat` without any parameters also goes on forever. Logically, the `repeat forever` loop is equivalent to `repeat while true`. Because you probably really don't want such a loop to go on forever, you need some way to terminate it. One is to use the standard Control-. (period) sequence for aborting. More likely, however, you would use `exit repeat` to stop the loop.

One place to use such a forever loop is when you wish the loop to respond to a variety of input signals. Here is a cute example.

Example 17:

1. Go to the Repeats card of your HyperTalk Bible stack and create a new button called `Forever`.
2. Enter the following handler in the button script:

```
on mouseUp -- Forever button
  put "harpsichord" into voice
  repeat forever
    play voice tempo 400 "c f f c"
    wait 40
    if the shiftKey is down then put "boing" into voice
    else if the optionKey is down then put "harpsichord" into¬
    voice
    if the mouseClick then exit repeat
  end repeat
end mouseUp
```

3. Click the Forever button.

Result:

The Macintosh plays a little four-note sequence repetitively in the harpsichord voice.

4. Hold the Shift key down long enough for the `if` statement to notice.

Result:

The Macintosh switches to the boing voice.

5. Hold down the option key. Wait, then click the mouse.

Result:

The Macintosh switches back to the harpsichord voice. When you click the mouse, the loop terminates.

You don't really need the `repeat forever` form. In Example 17, we could replace it with `repeat while` true or `repeat until false`. Or we could use `repeat until the mouseClick` and delete the final `if then` statement.

The *wait* Command

We already have discussed and used one form of the `wait` command: `wait for n`, where `n` is either ticks or seconds. Now let's examine the other forms of this command.

The *wait until* Command

The `wait until` command causes your script to wait until some condition becomes true. Figure 10-17 shows its syntax.

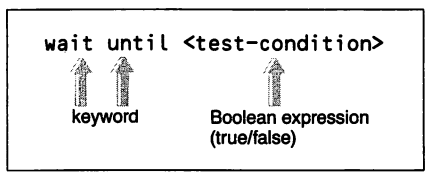


Figure 10-17. The `wait until` command syntax.

Here's an example in which you click a button but the action doesn't start until you press the Shift key:

Example 18:

1. Go to the Repeats card of your HyperTalk Bible stack and create a new button called `Wait Until`.
2. Put this script into the button:

```
on mouseUp -- Wait Until button
  wait until the shiftKey is down
  repeat until the shiftKey is up
    beep 2
    wait 20
```

continued

```
  end repeat
end mouseUp
```

3. Click the Wait Until button.

Result:

Nothing happens yet.

4. Hold down the Shift key for a while and then release it.

Result:

When you press the key, the beeping starts. When you release the key, the beeping stops.

The *wait while* Command

The `wait while` command causes the Macintosh to wait as long as a test condition remains true. Figure 10-18 shows the syntax.

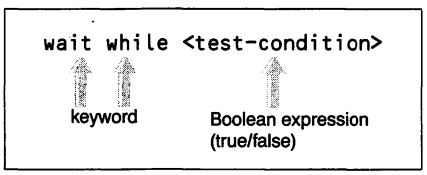


Figure 10-18. The `wait while` command syntax.

Let's see how it works.

Example 19:

1. Go to the Repeats card of your HyperTalk Bible stack and open the card script.
2. Enter the following text in the card script:

```
on mouseDown
  wait while the mouseH > 100
  repeat until the mouseH > 400
    beep 2
    visual effect checkerboard to black
    go to this card
  end repeat
end mouseDown
```

3. Position the mouse in the middle of the card and click the mouse button.

Result:

Nothing, yet.

4. Move the mouse to the left of the card.

Result:

The visual effect begins and continues until you move the mouse to the right side of the card.

The `wait while` command causes the script to wait until you move the mouse to the far left. This is one typical use for the `wait` command: delaying the start of a script until some particular action takes place.

You can use either the `wait while` or the `wait until` form for the same task; it's just a matter of selecting the proper test condition. For instance, in Example 19, we can replace `wait while the mouseH > 100` with `wait until the mouseH <= 100`.

Summary Script

Loops let you provide an interactive interface for a user. For instance, loops can be used to guide a user to proper input by rejecting wrong input. We can illustrate this approach through a guessing game. This also gives us a chance to use nested loops and to show how loops interact with `ask` and `answer` commands as well as with `if` statements.

Example 20:

1. Go to the Repeats card of your HyperTalk Bible stack and create a button called `Guess Again`.
2. Put the following script into the button:

```
on mouseUp -- Guess Again button
  answer "Let's play a guessing game!" with "Cancel" or "OK"
  if it is "Cancel" then exit mouseUp
  repeat while it <> "Cancel"
    put the random of 100 into mychoice
    ask "Guess a number from 1 to 100"
    if it = empty then exit mouseUp
    repeat until it = mychoice
      if it = empty then exit mouseUp
      else if it > mychoice then ask "Too high! Guess again!"
      else ask "Too low! Guess again!"
    end repeat
    answer "You got it! Play again?" with "Cancel" or "OK"
  end repeat
end mouseUp
```

3. Click the Guess Again button.

Result:

A dialog box (Figure 10-19) asks whether you want to play a guessing game.

4. Click OK.

Result:

Another dialog box (Figure 10-20) asks you to guess a number from 1 to 100.

5. Enter a number.

Result:

You're told (via a dialog box) that you're too high, too low, or correct. If you're wrong, you get to guess again. If you're right, you have the option of starting a new game. (Figures 10-21 and 10-22).

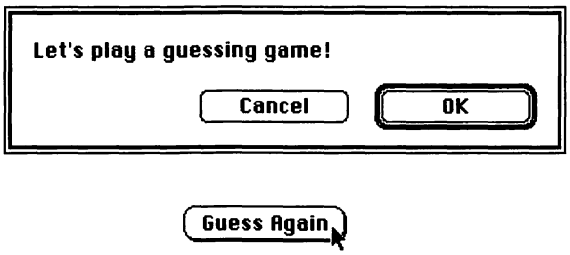


Figure 10-19. First dialog box.

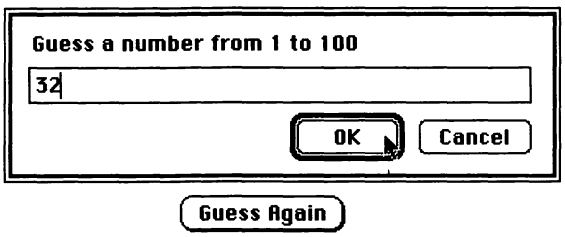


Figure 10-20. Second dialog box.

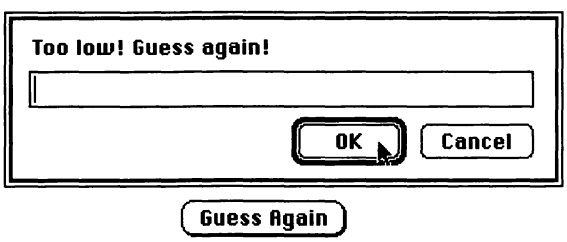


Figure 10-21. Third dialog box.

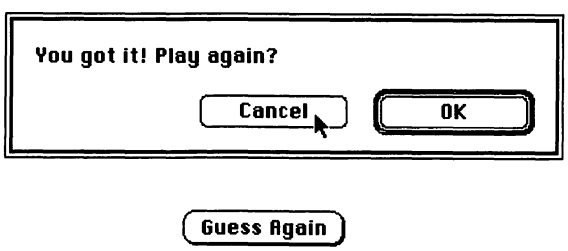


Figure 10-22. Fourth dialog box.

One feature worth noting is that the script lets you quit by selecting the Cancel button. This works differently for the `ask` and the `answer` commands. For the `answer` command in the script, Cancel was explicitly specified as a choice in the command. That means that clicking Cancel sets the global variable `it` to the string `Cancel`. But with the `ask` command, Cancel is a default button. Clicking OK sets the global variable `it` to the answer you entered, but clicking Cancel sets it to `empty`, not to the string `Cancel`. The script reflects these differences by comparing it to `Cancel` in one case and to `empty` in the other.

Review Questions

1. How many cycles do each of the following loops execute?

   a.
   ```
   repeat for 3 times
     beep
   end repeat
   ```
   b.
   ```
   repeat 7
     put "Ha " after the message box
   end repeat
   ```
   c.
   ```
   repeat with n = 1 to 5
     play "boing"
   end repeat
   ```
   d.
   ```
   repeat with m = 60 to 60
     play "boing" m
   end repeat
   ```
   e.
   ```
   repeat with m = 60 down to 70
     play "boing" m
   end repeat
   ```
   f.
   ```
   put 10 into n
   repeat while n > 0
      subtract 1 from n
   end repeat
   ```
   g.
   ```
   ask "Pick a number from 1 to 10"
   repeat while it <> 8
     answer "Wrong! Try again!"
   end repeat
   ```

2. How many beeps do each of the following scripts produce?

   a.
   ```
   repeat with n = 60 to 70
     if (n mod 5 ) = 0 then beep
   end repeat
   ```
   b.
   ```
   repeat for 4 times
     repeat for 3 times
       beep
     end repeat
   end repeat
   ```
   c.
   ```
   repeat with n = 1 to 4
     repeat with m = 1 to 3
       beep m
     end repeat
   end repeat
   ```

d.
```
repeat with n = 1 to 4
  repeat with m = 1 to 3
    beep n
  end repeat
end repeat
```

e.
```
repeat with n = 1 to 4
  repeat with m = 1 to 3
    beep m + n
  end repeat
end repeat
```

3. Which commands discussed in this chapter use true-false expressions?
4. What's the difference between `exit repeat` and `next repeat`?
5. What causes each of the following loops to terminate?

a.
```
repeat with note = 60 to 70
   play "harpsichord" note
end repeat
```

b.
```
repeat while the shiftKey is up
  beep
end repeat
```

c.
```
repeat until the mouseClick
  beep
  if the optionKey is down then next repeat
  beep
  if the mouseH > 500 then exit repeat
end repeat
```

6. What is the purpose of `end repeat`?
7. a. Can a `repeat with` loop do the work of a `repeat while` loop?
   b. Can a `repeat while` loop do the work of a `repeat with` loop?
   8. Can the limits in a `repeat with` loop be variables? functions? containers? properties?

Exercises

1. Write a button script that asks you for a lower-limit integer and an upper-limit integer. Have the script sum all the integers from the lower-limit to the upper-limit and report the answer. Have the script provide default values for the two limits if the user doesn't enter any.
2. Write a button script that displays in the Message box one by one in reverse order (with a half-second pause between words) the words in a field.
3. Write a script that plays notes in the boing voice until the Shift key is pressed. Have each note be chosen randomly in the range 50 to 80.
4. Write a script that plays repeated sequences of ascending notes until the Shift key is pressed. Have the tempo for each note sequence be selected randomly. Have the starting note and the number of notes per sequence be chosen randomly.

Chapter 11

Strings

Quick View

Commands

select

Keywords

empty
space
tab
formfeed
linefeed
return
character, char
word
line
item

Functions

the length
offset()
the charToNum
the numToChar
the number of <chunks>
the selectedText
the selectedField
the selectedLine
the selectedChunk

Operators

contains
is within
&
&&

Container

the selection

Concepts

Unquoted strings
Numbers and strings
Joining strings
Text constants
ASCII codes and functions
Converting numbers to text
Chunks: units of text
Chars, words, items, lines
Ranges
Counting functions
Strings within strings

Chapter 11

Strings

String is a word with many meanings and uses. A G string means something quite different to a guitarist than it does to an exotic dancer. The enormous cosmic strings about which astronomers speculate are quite different from the ten-dimensional strings that some physicists think may underlie all of existence. (Life can be confusing to the astrophysicist who also is a guitarist and an exotic dancer.) In HyperTalk, a string is a series of characters considered as a unit. For example, the text in a text field is a string. The command you type into the Message box is a string. The sentence you are reading this very moment is a string, as is each individual word in this sentence. Indeed, this whole chapter is a string.

Strings are fundamental to HyperCard and HyperTalk. Text data and numeric data are stored as strings. Scripts are strings. You use strings to communicate with HyperTalk; for instance, the `ask` command displays and reads strings. It's only natural, then, that HyperTalk provides several string-related commands, operators, and functions.

In this chapter, we examine these tools to see how they can help you. You learn how to piece several strings into one. This is handy when you combine information from several sources. For instance, you may take a name from one field, do a calculation on a number from another field, and then incorporate the name of the field and the result of the calculation into a sentence. You also learn how to extract individual words, lines, and other substrings from a longer string. First, however, let's review the basics.

String Basics

In HyperTalk, a string can be a string constant (`"Apple"`), a value of a container (a text field or a variable), a property (the name of a button), or even the output of a command or function. The most elementary form is the string constant, also called a *literal,* which normally is expressed by placing the string into quotes. For example, `"twelve bonnie lasses"` is a string constant. Only the characters inside the quotes

are part of the string; the quotes mark the beginning and end of the string but are not part of it.

What can you do with string constants? You can use them in commands. We've already done so with the `ask` and `answer` commands. For example, the command `answer "Shall we continue"? with "Yes" or "No"` uses three string constants to indicate what will be displayed by the dialog box. You can use string constants in comparisons, as in `if it is "No" then exit mouseUp`. Also, you can assign strings to containers. Let's look at a few simple examples of that:

Example 1:

1. Go to your HyperTalk Bible stack and create a new card called `Strings`.
2. Create a card field called `Samples` for the new card; check the Rectangle box in the field dialog box so that the field is easily visible.
3. Open the Message box and enter

```
put "Howdy!" into card field "Samples"
```

Result:

`Howdy!` appears in the card field.

This shows that we can use the `put` command to place a string into a field. Also, note that we use the string `Samples` to identify the name of the field.

4. In the Message box, enter

```
put "Hamlet" into hottub
```

5. Enter `hottub` in the Message box.

Result:

`Hamlet` appears in the Message box.

As promised, you can put a string (`Hamlet`, in this case) into a variable. Then, you can use the variable (`hottub` here) the same ways you use a string literal.

6. Enter

```
put hottub after card field "Samples"
```

in the Message box.

Result:

The field now holds `Howdy!Hamlet`.

By using the `put after` form, you can add to the string already in the field. This is an example of *concatenating*, or joining, strings.

7. Enter

```
put Howdy before card field "Samples"
```

in the Message box.

Result:

The field now holds `HowdyHowdy!Hamlet`.

First, by using the `put before` form, you can add material to the beginning of a string. Second, we omitted the double quotes around `Howdy`, and the command still worked! Can you always expect such good fortune? Let's push our luck with another example.

8. Enter

```
put come here after card field "Samples"
```

in the Message box.

Result:

A dialog box complains `can't understand arguments to put`.

This time, omitting the quotes caused trouble. We answer the question of why in a moment or two.

9. Click Cancel in the dialog box, clear the Message box, and enter

```
put ", the Prince of Denmark" after hottub
```

in the Message box.

10. Enter `hottub` in the Message box.

Result:

The Message box displays `Hamlet, the Prince of Denmark`. Spaces and commas also count as characters, and now `hottub` contains a whole phrase as a string.

11. Enter

```
put "Ophelia" into hottub
```

in the Message box, and then enter `hottub` in the Message box.

Result:

The Message box displays `Ophelia`.

When you use `put into` to place a string into a container already holding a string, the new string replaces the old. To get `Ophelia` into `hottub` with `Hamlet, the Prince of Denmark`, you need to use the `put after` or the `put before` form.

An Unquoted String Can Be a Troublesome Thing

Going back to Steps 6 and 7, just when can you omit the quotes? First of all, you shouldn't omit them, because they serve to identify the string clearly in a command or script. You'll probably omit them anyway, however (through laziness, by accident, or in search of adventure), so let's see what's going on. Compare these two commands from Example 1:

```
put hottub after card field "Samples"
put Howdy before card field "Samples"
```

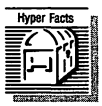
In the first command, `hottub` is a HyperTalk variable; in the second example, `Howdy` is a string. Because `Howdy` is not in quotes, HyperTalk should interpret `Howdy` to be the name of a variable, just as `hottub` is, but Hypertalk notices that `Howdy` has not yet been used as a name for a variable. Its inner rules assume that you were lazy or forgetful and that `Howdy` really is a string.

String Characters

The characters in a string aren't limited to letters of the alphabet. Punctuation marks such as quotes, commas, and exclamation points are characters, as is the space character. The digits 0–9 are characters. Some characters are nonprinting and act more like instructions. For example, there is a linefeed character that advances the cursor to the next line. Like most computers, the Macintosh supports a standard set of 128 characters called the ASCII character set, and the characters we've mentioned are part of that set. The Macintosh also supports additional characters. We return to the ASCII character set later. Meanwhile, the important point is that strings can contain any of these sorts of characters.

Numbers and Strings

HyperTalk stores numbers as strings. For instance, if you give the command `put 2010 into hal`, HyperTalk stores the string `"2010"`. That is, it stores the four characters `2`, `0`, `1`, and `0` rather than the numeric value `2010`. Fortunately, HyperTalk is smart enough to convert the string to the corresponding numeric value when you say `put hal + 20 into the message box`.



Ampersand Trick
When you want to join two strings, a space can be added between them by using two ampersands (&&). One ampersand joins the strings without an intervening space.

Joining Strings: the & and && Operators

Frequently, you'll need to construct longer strings from smaller ones. Perhaps, you'll want to insert someone's name into a stock phrase: `This is your lucky day, Phillipe Gates`, or perhaps your script has calculated a mortgage payment, and you want to display the answer as part of a sentence: `Your monthly payment is $1234.21`. Perhaps you want a script to put together a command to be run by the `do` command. This joining process is called concatenation, and HyperTalk makes it easy. First, let's discuss the mechanics of concatenation.

The HyperTalk operators `&` and `&&` are the tools you need to join strings. The `&` operator joins strings so that the first character of the second string immediately follows the last character of the first string. However, the `&&` operator inserts a space character between the two strings. The syntax for these operators is shown in Figures 11-1 and 11-2. In Figures 11-1 and 11-2, the optional spaces are not included in the final string.

A short session with the Message box shows how these operators work:

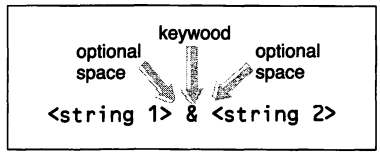


Figure 11-1. The & operator syntax.

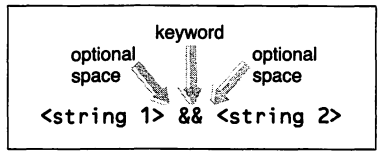


Figure 11-2. The && operator syntax.

Example 2:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Enter

```
"the" & dogs"&"hip"
```

Result:

The Message box displays `thedogship`.

The three strings `the`, `dogs`, and `hip` are joined into the single string `thedogship`. Note that it makes no difference whether or not you surround the & with spaces.

3. Enter

```
"dogs"&&"hip"
```

Result:

The Message box displays `dogs hip`.

This time the resulting string does contain a space, because that is how the && operator works.

These examples show how the & and && operators work, but there's not much point to using these operators with ordinary string literals, because you could as easily type `"thedogship"` or `"dogs hip"` in the first place. What makes these operators worthwhile is using them with containers, functions, properties, and with some HyperTalk constants. Here's an example using containers.

Example 3:

1. Go to the Strings card in your HyperTalk Bible stack and create a button called `Concatenate`.
2. Enter the following script in the button:

```
on mouseUp -- Concatenate button
  ask "What's your name?"
```

continued

```
    put it into name
    ask "What's your favorite animal?"
    put it into animal
    answer "I once knew an" && animal && "named " & name & "!"
end mouseUp
```

3. Click the Concatenate button.

Result:
A dialog box requests your name.

4. Enter your name in the dialog box.

Result:
A dialog box requests your favorite animal.

5. Enter the name of your favorite animal.

Result:
A dialog box displays a message composed from script strings combined with your input. Figure 11-3 shows a sample.

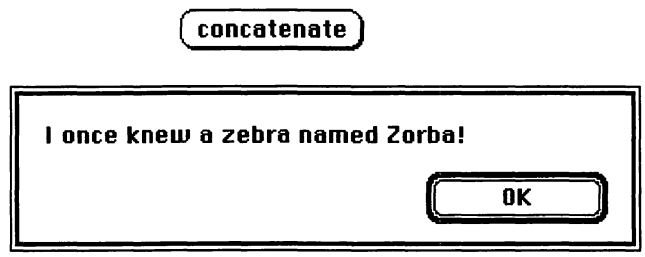


Figure 11-3. Sample output.

The key portion of the script is this concatenated string:

```
"I once knew an" && animal && "named " & name & "!"
```

It combines string constants (the items in quotes) with the string variables `animal` and `name` to form the final message. Notice that we used two ways to put spacing into the string. For `animal`, we used the `&&` operator to provide the spacing. For `name`, we used a space character at the end of the string `"named "` to provide spacing. We didn't want any spacing between `name` and `"!"`, so we had to use the `&` operator there.

The concatenation operators can be used to incorporate certain special characters into strings. We look at that topic next.

HyperTalk Text Constants

HyperTalk has six built-in constants to represent special constants. Table 11-1 lists them and their definitions.

Table 11-1. HyperTalk Constants

HyperTalk Constant	Meaning
empty	The empty string: ""
space	The space character: " "
quote	The quote character: " "
return	The return character: go to the beginning of the next line or (for some printers) go to the beginning of the same line
tab	The tab character
linefeed	The linefeed character: go down one line (printer)
formfeed	The formfeed character: go to next page (printer)

We've already used `empty` often. For example, in chapter 9, "Ifs," we used the construction `if it is empty` to check whether a user responded to an ask dialog box by selecting Cancel. That response sets `it` to the `empty` string. The `empty` constant also can be used to clear a container. For instance, the command `put empty into field 1` sets the contents of that field to the `empty` string.

The space character has many uses. For example, you can use it with `offset` (discussed later in this chapter) to find the first space in a string. If you want to separate two concatenated strings with two spaces rather than one, you can type `string1 && space & string2`. This is equivalent to `string1 & " " & string2`.

You can use the quote character to include quote marks in a string. The next example assumes that you haven't deleted the card field Samples which you created for Example 1.

Example 4:

1. Go to the Strings card of your HyperTalk Bible stack.
2. Enter

   ```
   put quote&"Pig-lips"&quote&&"is here!" in card field "Samples"
   ```

   in the Message box.

Result:

The field reads `"Pig-lips" is here!`

The quote constant is replaced by " in the final text.

Note that the quote constant has to be outside the quote marks. If you put it inside the quotes, it's interpreted as the word *quote*.

The effect of the return character depends on how you use it. Within HyperCard text fields, it moves the cursor to the beginning of the next line.

Example 5:

1. Go to your HyperTalk Bible stack and the Strings card and create a new card field called `Friends`.
2. Make the new field a rectangle field so that it can be seen easily and move the Friends and Samples fields so that they don't overlap.

3. Now enter

```
put "Fifi LaBozo" into card field "Friends"
```

in the Message box.

4. Next, enter

```
put "Lefty Dorkus" after card field "Friends"
```

in the Message box.

Result:

The first line of the field reads as follows:

```
Fifi LaBozoLefty Dorkus.
```

The `put after` command works similarly to the & operator; it attaches the new string to the end of the old.

5. In the Message box, enter

```
put "Fifi LaBozo" into card field "Friends"
```

6. Now enter

```
put return & "Lefty Dorkus" after card field "Friends"
```

in the Message box.

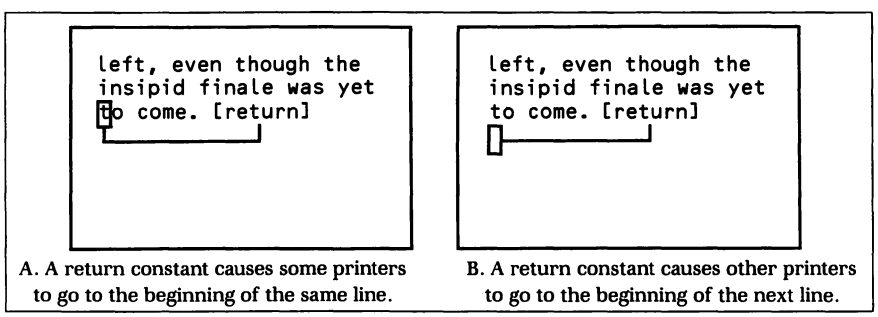
Result:

The field now reads as follows:

```
Fifi LaBozo
Lefty Dorkus
```

The return constant between the two names causes HyperTalk to move to the beginning of the next line in the field. Thus, the return constant provides a simple way to start new lines when you use `put` to construct text.

Sending the return constant to a printer may have a different effect. The literal meaning of this character is "Move the print head to the beginning of the current line." However, many printers automatically insert a linefeed character when they are sent a return in order to position the print head at the beginning of the next line. See Figure 11-4.



A. A return constant causes some printers to go to the beginning of the same line.

B. A return constant causes other printers to go to the beginning of the next line.

Figure 11-4. Sending a return constant to a printer.

The tab constant represents the character produced by the Tab key. In the HyperCard context, pressing the Tab key selects the text of each field in a card in turn. Here's an example that shows you can use tab in a script to the same effect:

Example 6:

1. Go the Samples card of your HyperTalk Bible stack and create a new button called `Tab`.
2. Put the following script into the Tab button:

```
on mouseUp -- Tab button
  repeat until the mouseClick
    type tab
  end repeat
  beep
end mouseUp
```

3. Click the Tab button, watch a moment, and then click the mouse again.

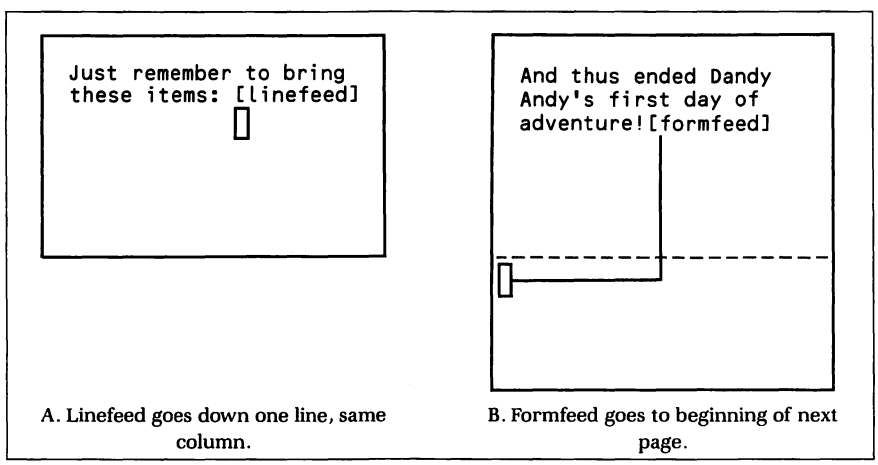
Result:

First the Sample field is highlighted, then the Friends field, then Sample, and so on, until you click the mouse again. Then the script beeps and terminates.

The command `type tab` is equivalent to pressing the Tab key from the keyboard, so this script works the same way as typing the Tab key repeatedly.

The tab character also is used often in external database files to separate one data field from another.

The formfeed and linefeed characters typically are used with the HyperTalk `write` command to place these characters into a file. The formfeed character instructs a printer to advance to the top of the next page; the linefeed character tells a printer to move down one line in the same column. See Figure 11-5. By placing these characters into a file, you control how the file is printed.



A. Linefeed goes down one line, same column.

B. Formfeed goes to beginning of next page.

Figure 11-5. Formfeed and linefeed characters.

Printing a Portion of Your Script
When HyperTalk is in the script editor, you can select a range of script, and when the Print button is pressed, only the portion you selected is printed.

ASCII Code and the ASCII Functions

While we are on the topic of characters, let's discuss how HyperTalk and the Mac handle them. We said that a string is stored as a series of characters, but how is a character stored? Essentially, computer memory is set up to store numbers, so the obvious answer is to develop a numeric code to represent characters. The most widely-used code for this purpose is the ASCII (American Standard Code for Information Interchange) code we mentioned earlier, and that is the code the Mac uses. In this code, the numbers 0 through 127 represent a standard character set, called the ASCII set, so the string `fun` is represented by storing in succession the numeric codes for the characters f, u, and n, which happen to be 102, 117, and 110. In addition, the Mac uses the numbers 128 through 255 to represent additional characters. The exact characters in this extended set depend on the font, and not all of the values may be used.

To tell whether a value of 102 in memory really is just 102, or whether it represents the character f, a program has to keep track of which kind of data is stored where. HyperCard takes care of those details for you.

Sometimes, you may want to know the code for a particular character. For example, you might want to check whether a particular letter is uppercase. In general, recall, HyperTalk doesn't distinguish between upper- and lowercase letters, but the two cases do have different ASCII codes. You may want to represent some character with no simple keyboard equivalent by using its ASCII code. HyperTalk provides two functions to give you access to ASCII codes. The function `charToNum()` (see Figure 11-6 for syntax) returns the code for a given character, and the function `numToChar()` (see Figure 11-7 for syntax) displays the character that corresponds to a given code number. Note that each function has two equivalent forms.

```
charToNum(<character>)
the charToNum of <character>
```

Figure 11-6. The `CharToNum()` syntax.

```
numTochar(<number>)
the numToChar of <number>
```

Figure 11-7. The `NumToChar()` syntax.

Example 7:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Enter

```
charToNum("A")
```

Result:

The number 65 appears in the Message box. This is the ASCII code for the character A.

3. Enter

```
charToNum of "a"
```

Result:

The number 97 (ASCII for a) appears in the Message box.

As promised, a and A have different ASCII codes. If you type `"a" = "A"` into the Message box, HyperTalk says true, but if you type `charToNum("a") = charToNum("A")` into the Message box, you get false. This is one way to discriminate between lower- and uppercase.

4. Enter

```
numToChar(66)
```

Result:

The character `B` appears in the Message box.

This is the letter with the code 66. Note that this is 1 greater than the code for A.

Let's use what we've learned to create a table of ASCII values. Because the resulting field will take up most of a card, create a new card.

Example 8:

1. Go to your HyperTalk Bible stack and Strings card.
2. Type Command-N to create a new card; name it `ASCII`.
3. Create a new field. Select the scrolling field format. Click the font button and verify that the field font is the standard Geneva 12-point font. Size the field to nearly full card. Leave room at the bottom for a button to show.
4. Create a button called `ASCII Code`, enter the following script in it, and place the button below the field:

```
on mouseUp -- ASCII Code button
  if the number of card fields < 1 then
    answer "Sorry, you need a card field"
    exit mouseUp
  end if
  ask "Starting ASCII code value?"
  if it is empty then exit mouseUp
  put it into start
  ask "Final ASCII code value?"
  if it is empty then exit mouseUp
  put it into finish
  put empty into card field 1
  set the numberFormat to "000"
  repeat with n = start to finish
```

continued

```
      put n && "=" && numToChar(n) & space & space after card¬
      field 1
      if (n - start) mod 8 = 7 then put return after card field 1
    end repeat
  end mouseUp
```

5. Click the ASCII Code button and enter 33 as the start value and 127 as the final value.

Result:

The ASCII codes and corresponding characters are placed into the field, as shown in Figure 11-8.

We start with the value 33 because the preceding 32 characters are nonprinting.

Note that the script exits if there is no card field or if you select Cancel in either dialog box. Otherwise, the script runs a loop to display the codes and characters. Note how we used the concatenation operators and the text constant space to construct the output string:

```
n && "=" && numToChar(n) & space & space
```

Here n is the code number and `numToChar(n)` is the corresponding character. If, say, n is 65, the final string is `"65 = A "`. The spaces serve to separate this string from the next.

The second line in the loop starts a new line every 8 strings:

```
if (n - start) mod 8 = 7 then put return after card field 1
```

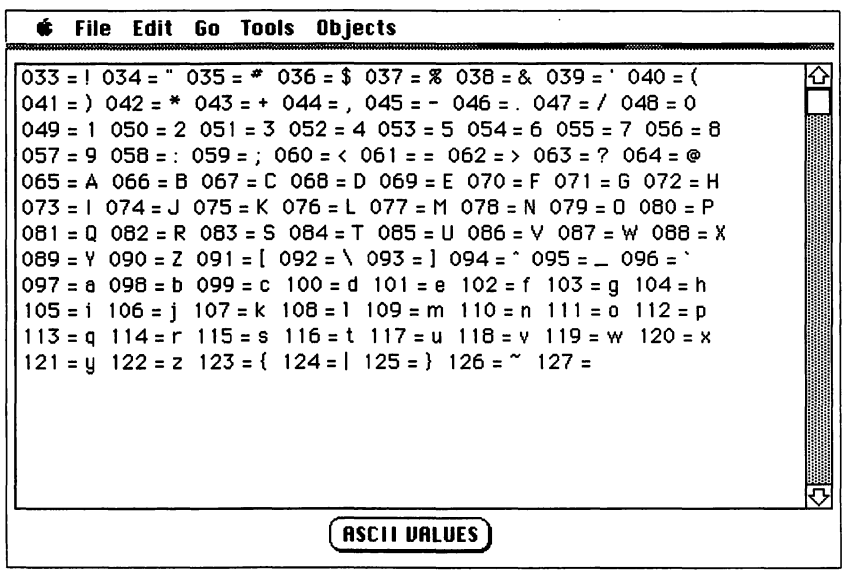


Figure 11-8. ASCII codes.

Suppose `start` is 33 and `n` is 39. Then, `n -start` is 6, `6 mod 8` is 6, and the test condition is false. But if `n` is 40, `n - start` is 7, and the test condition is true. This causes the script to put a return character into the field, starting a new line. The next time the test is true is when `n` is 48. Then `n - start` is 15, and 15 mod 8 is 7, so every time `n` increases by 8, another new line is started.

The command `set the numberFormat to "000"` causes HyperTalk to display numbers in a three-digit format, so 40 is shown as 040. That helps even out the line lengths. However, because the Geneva font is a proportional one, not all characters have the same widths. Although each line has the same number of characters, the line lengths differ. To get a nicely aligned display, open the field dialog box, click the font button, and select Monaco 9. This font gives all characters the same width. To investigate the extended codes of a font, use 128 and 255 as the lower and upper limits.

Text Units: HyperTalk Chunks

HyperTalk breaks text into several different units. Suppose that you have some text in a container, such as a variable or a field. You can treat the whole text as a unit, or you can access different "chunks" of it. The possible chunks are the "word", the "line", the "character" (or "char", for short), and the "items."

Basic Chunks

There are the basic chunks, and there are combinations and ranges of these chunks. Look at the basic elements first.

A character is just that: a single character.

A word is any series of characters whose beginning and end are marked by *white space* (spaces, tabs, and returns) or by the beginning or end of the container. Thus the string `"Drop that #$^!@ hat!"` has four words. Note that the fourth word, `hat!`, includes the exclamation point. That's because there is no space between the exclamation point and hat. There is a catch to this definition of a word: A quoted phrase within a string counts as a single word.

A line is marked by the return character (or the end of the container), but doesn't include the return character. That is, the first line of a field is everything up to the first return character. The second line is everything between the first and second return characters, and so on to the end of the field.

Items are units of text separated by commas. The comma itself is not part of an item; it just marks the boundary between items.

You can indicate individual chunks in a text by numbering them. Let's see how that works:

Example 9:

1. Go to the Strings card of your HyperTalk Bible stack.
2. Add three more lines to the card field Friends so that it reads as follows:

```
Fifi LaBozo
Lefty Dorkus
Treakle, Nert
Truhart, Tess
"Big Feet" Ben
```

   You can do this by clicking the mouse just after `Dorkus` thereby placing the insertion point there. Then, press the Return key and type the final three lines.
3. Enter

```
put card field "Friends" into card field "Samples"
```

   in the Message box.

Result:

The contents of Friends is duplicated in Samples.

This illustrates treating the whole text as a unit.

4. Enter

```
put line 2 of card field "Friends" into card field "Samples"
```

   in the Message box.

Result:

Samples displays `Lefty Dorkus`.

Lines are identified by number. Line 2 is the one following the first return character and preceding the second.

5. Enter

```
put line 1 of card field "Friends" after card field "Samples"
```

   in the Message box.

Result:

Samples displays `Lefty DorkusFifi LaBozo`.

This demonstrates that line 2 did not include the return character that separates lines. If it did, there would have been an invisible return character after Dorkus in Samples, and the addition would have gone on the next line.

6. Enter

```
put word 2 of card field "Friends" into card field "Samples"
```

   in the Message box.

Result:

Samples displays `LaBozo`.

7. Enter

```
put char 6 of card field "Friends" into card field "Samples"
```

   in the Message box.

Result:

Samples shows `L`.

8. Enter

```
put item 2 of card field "Friends" into card field "Samples"
```

in the Message box.

Result:

Samples looks like this:

```
 Nert
Truhart.
```

Everything between the first and second commas, including the space before Nert and the return after Nert, comprise the second item.

Composite Chunks

You can use any of the chunks as parts of larger chunks. You can specify a particular letter in a word or a particular word in a line. For example, you can use expressions like `char 4 of word 2 of card field "Friends"` or `char 4 of word 2 of line 3 of card field "Friends"`. To see what these expressions mean, start at the right end of the expression and work to the left. For instance, `card field "Friends"` is the field we just worked with. Line 3 of card field "Friends" is Treakle, Nert. Word 2 of line 3 card field "Friends" is Nert. Char 4 of word 2 of line 3 of card field "Friends" is t. Finally, word 1 of line 5 of card field "Friends" is "Big Feet" because a phrase inside double quotes in a string counts as a single word. The spaces inside the quotes are ignored for the purposes of determining where a word begins or ends.

Ranges

You also can have ranges, such as `char 6 to 10`, meaning characters 6 through 10. Look at some examples now.

Example 10:

1. Go to the Strings card of your HyperTalk Bible stack.
2. Check that the Friends field is as described in Example 9.
3. Enter

```
put char 6 to 20 of card field "Friends" into card field "Samples"
```

in the Message box.

Result:

The Samples field looks like this:

```
LaBozo
Lefty Do
```

Because the return character was included in the range, we get two lines.

4. Enter

```
put word 4 to 6 of line 2 to 4 of card field "Friends" into card
field "Samples"
```

in the Message box. (This takes some faith on your part, because the command is longer than the box.)

Result:

Samples looks like this:

```
Nert
Truehart, Tess
```

So we can use combinations of ranges, too!

One important point about ranges is that all characters between the first and last element of the range are included. So although spaces and return characters normally are not part of a word, those found between word 4 and 6 count as part of the range word 4 to 6.

Chunks as Destinations

So far, we have used chunks to specify a source of text to be copied. They also can be used as destinations. Use the form `put <text> before <chunk>` to place material before a particular chunk, and use the form `put <text> after <chunk>` to put it after the chunk. Finally, if you use the form `put <text> into <chunk>`, the `<chunk>` is replaced with `<text>`. With all these forms, of course, you have to specify the container holding the chunk. Let's try a few.

Example 11:

1. Go to the Strings card of your HyperTalk Bible stack.
2. Check that the Friends card is as described in Example 9.
3. Open the Message box and enter

   ```
   put "ley" after line 3 of card field "Friends"
   ```

Result:

The third line now reads `Treakle, Nertley`.

4. Enter

   ```
   put "ena" into char 11 of card field "Friends"
   ```

   in the Message box.

Result:

The first line now reads `Fifi LaBozena`.

Note that we can replace one character with several; the `<text>` doesn't have to be the same size as the `<chunk>` it replaces.

5. Clear the Samples field by tabbing to it and pressing the Backspace key.
6. Enter

   ```
   put "HyperSnark" into line 3 of card field "Samples"
   ```

in the Message box.

Result:

`HyperSnark` appears on the third line of the otherwise empty field.

Because the field was empty, this command caused two return characters to be placed into the field before HyperSnark, making it the third line. Similarly, HyperTalk supplies extra commas if you put something into item 4 in an empty field. However, there is no similar effect for words or characters.

Strings, Numbers, and *numberFormat*
Whenever HyperCard performs an operation that uses strings as its input, such as concatenation, the program first uses the current setting of the `numberFormat` property to represent any numbers as strings. The result is a string, not a number.

Counting Functions

If you work with text, you'll often find it useful to know how long a string is. Perhaps you want to run a `repeat` loop that processes each character in a string. Knowing the string length tells you how many loop cycles you need. Perhaps you want to know the exact length of a block of text you just wrote. HyperTalk can tell you the answer through its counting functions `length()` and `the number`. The `length()` function is the simpler of the two, so let's start with it.

The *length()* Function

The `length()` function takes one argument, a string, and it returns the number of characters in that string. You have your choice of two forms, as seen in Figure 11-9.

```
length(<source string>)
the length of <source string>
```

Figure 11-9. The `length()` function syntax.

Let's take another trip to the Message box and try this function:

Example 12:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Enter

```
the length of "Monty Python"
```

Result:

The Message box displays 12.

There are twelve characters in Monty Python. Note that the space counts as a character.

You can use `length()` with containers, too.

Example 13:

1. Go to your HyperTalk Bible stack and to the ASCII card. (We assume it still contains a nonempty field from Example 7. If it doesn't, remedy the situation.)
2. Open the Message box and enter

```
length(card field 1)
```

Result:

If you have the same field we do, the Message box shows 1168.

Recall that we have to provide the adjective `card`; otherwise, HyperTalk assumes we mean a background field.

Also, you can use `length` with the various chunk expressions, as in `the length of line 2 of card field "Friends"`.

The *number of* Function

You've already met the function `the number` in Example 6 of chapter 4, "Containers and Values." There we used it to see how many cards a stack had (`the number of cards`). This function also has text-measuring capabilities. You can use it to count the number of characters, the number of words, the number of items, and the number of lines in a text source. Of course, you have to tell the function which kind of thing it's supposed to count. Figure 11-10 shows the syntax.

```
the number [of] <components> in <container>
```

Figure 11-10. The `the number of` function syntax.

The counting units are based on the chunk units you just learned about. So `chars` counts text in `char` units. Thus, `the number of characters in card field 1` is the same as `length(card field 1)`, except that the former requires more typing. Similarly, `words` counts in `word` units, `lines` counts in `line` units, and `items` counts in `item` units.

Example 14:

1. Go to the Strings card of your HyperTalk Bible stack and open the Message box.
2. Clear the Samples field.

3. Enter the following text in the field. Do not use the return key; instead, let the text wrap itself automatically:

```
Then did mighty-thewed Haxar thump the table mightily, yelling for
more ale, then singing his favorite song, "Haxar the Magnificant."
```

Result:
The card field should look like Figure 11-11, assuming you haven't resized it.

```
Then did mighty-thewed Haxar
thump the table mightily,
yelling for more ale, then
singing his favorite song,"
Haxar the Magnificant."
```

Figure 11-11. Field text for Example 14.

4. Open the Message box and enter

```
the number of characters in card field "Samples"
```

Result:
The Message box displays `135`. (Spaces and punctuation are counted.)

5. Enter

```
the number of words in card field "Samples"
```

in the Message box.

Result:
The Message box displays `18`.

The hyphenated phrase `mightily-thewed` counts as one word, as does the quoted phrase `"Haxar the Magnificant"`.

6. Enter

```
the number of items in card field 2
```

Result:
The Message box displays `4`, because the three commas divide the field into 4 items.

7. Enter

```
the number of lines in card field "Samples"
```

Result:
The Message box displays `1`!

HyperCard wraps the line to make it fit in the field, but because you didn't press the Return key, there are no return characters in the field. (That assumes you patiently followed our instructions.) If you resize the box, for example, the apparent line breaks adjust to conform to the box size. This is done without any return characters being added to the text. By HyperTalk logic, because the text contains no re-

turn characters, it consists of one line. To get lines in the text, use the Return key when you enter text or insert Return keystrokes later.

You can apply `the number` and `length()` to any string, including parts of a text field. For example, you can specify particular lines and words in a text field, as in `length(line 2 of card field "Samples")` or `the number of words in line 3 of card field "Samples"` or even `length(line(the number of lines in card field "Samples") of card field "Samples")`.

The last example was a bit complicated, so look at it more closely. The expression `the number of lines in card field "Samples"` evaluates to 5, because the field has 5 lines. This makes the entire expression read `length(line 5 of card field "Samples")`. The power of the original expression is that it gives the number of characters in the last line of a field, regardless of the number of lines; it figures out the number of the last line for you.

Word Counting with a Button

When working with text, one of your most basic needs is for a word counter. For example, when you write a term paper or report, you usually want to know how many words, lines, and characters you've written. In general, the hardest thing about writing a word counter is writing the code for recognizing what a word is. But HyperTalk already has a built-in concept of a word, so you are spared that labor. Indeed, the HyperTalk string counting functions make designing a word-counting button a snap.

Example 15:

1. Go to the Strings card of your HyperTalk Bible stack.
2. Create a background button called `Word Counter`.
3. Enter the following script in the button:

```
on mouseUp -- Word Counter button
  put the number of card fields into numfields
  if numfields > 0 then
    ask "Which field? Maximum value is" && numfields with 1
    put it into fieldval
  else exit mouseUp
  if fieldval > numfields or fieldval < 1 then
    answer "I can't accept that value. Bye."
    exit mouseUp
  else
    put the length of card field fieldval into charcount
    put the number of words in card field fieldval into wordcount
    answer "Card field" && fieldval & ":" && wordcount &&¬
    "words;" && charcount && "chars"
  end if
end mouseUp
```

4. Click the Word Counter button.

Result:

A dialog box asks you to select a field.

5. Select a field by number.

Result:

An answer box (see Figure 11-12) tells you how many words and characters are in the field.

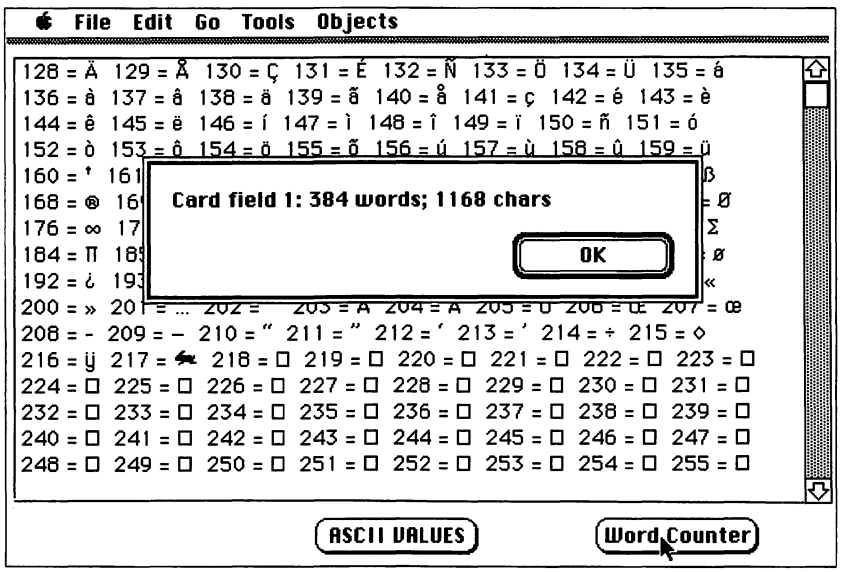


Figure 11-12. Word count.

We incorporated several `if` statements in the script to make it run more smoothly and to keep them fresh in your mind. Also note how the `&` and `&&` operators make it possible to incorporate variables into the strings displayed by the `ask` and `answer` commands.

Strings Within Strings

Is Philbo Phrip in your customer stack? Which line in scrolling field 3 has the plutonium data? Did the user's response include the secret word? Where did you put that Thrill button? The more HyperCard material you create, the more need you have for tools to find things. Of course, HyperTalk obliges. HyperCard itself has a `find` command, which searches a stack or field for a given phrase. HyperTalk also adds two operators (`contains` and `is in`) that determine whether or not a given string is found within another. It also provides a function (`offset()`) that tells where in a string a substring is found. With these facilities, you can do things like find all the address lines in a field containing a particular city or zipcode. You can scan input for the presence of certain keywords.

Testing for Content: *contains* and *is in*

The `contains` and `is in` operators are logical, or Boolean, operators that test whether one string contains the other. The syntax is simple and is shown in Figures 11-13 and 11-14.

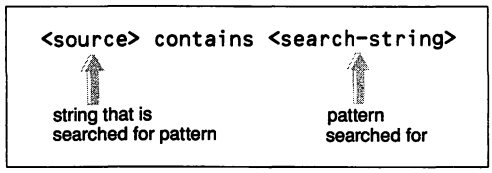


Figure 11-13. The `contains` operator syntax.

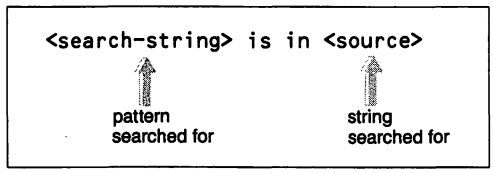


Figure 11-14. The `is in` operator syntax.

The purpose of each operator is to test whether `<search-string>` is in `<source>`. If so, the expression has the value `true`; otherwise the expression is `false`. The two forms really are different ways of saying the same thing. That is, `"see spot run" contains "spot"` is equivalent to `"spot" is in "see spot run"`.

In general, the `<search-string>` and `<source>` can be any sort of container or expression (including chunk expressions) that evaluates to a string. To see how these operators work, begin by using simple strings.

Example 16:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Enter

   ```
   "see spot run" contains "spot"
   ```

   in the Message box.

Result:

The Message box displays `true`.

3. Enter

   ```
   "Spot" is in "see spot run"
   ```

Result:

The Message box displays `true`.

As we've seen before, HyperTalk ignores the distinction between upper- and lowercase letters.

4. Enter

```
"the" is in "other"
```

Result:

The Message box displays `true`.

These operators don't require that `<search-string>` stands apart as a separate word (or words) in `<source>`; `<search-string>` can be embedded in a word, as `the` is in `other`. Similarly, the expression `"the day of the hacker" contains "the hack"` is true. On the other hand, `"boo" contains "boot"` is false, because `<source>` must contain the whole substring, not just part of it.

More typically, one or both strings (source or search) are a variable or a field or part of a field. Look at a modest database example. Suppose that you use a card field to hold the names and grades shown in Figure 11-15 and that you want to know who got some form of A (A+, A, or A-) on the fourth quiz. In HyperTalk, the presence of commas divides each field into five items, so the fifth item is the grade for the fourth quiz. Because the grade may contain a plus or minus, you can't just use the equality operator to compare the item to A. By using `is in`, however, you can check to see whether the fifth item contains an A character.

```
Jason Blimp, A-, A, B+, A
Margo Dandiform, B+, A, B-, A
Kent Fungo, B, C+, D, F
Narcissa Herculon, C, C-, B, A
Alonzo Wunderbar, F, A+, C, A-
```

Figure 11-15. A field of grades.

Here is an example implementing that approach.

Example 17:

1. Go to the Strings card of your HyperTalk Bible stack. Create a new card field called `Grades`.
2. Enter the text from Figure 11-16 in the field.
3. Create a button called `Find A's` and give it the following script:

```
on mouseUp -- Find A's button
  put number of lines in card field "Grades" into lines
  put empty into names
  repeat with lnum = 1 to lines
```

continued

```
      if "A" is in item 5 of line lnum of card field "Grades" then
        if names is empty then
          put item 1 of line lnum of card field "Grades" into names
        else put ", " & item 1 of line lnum of card field "Grades"¬
        after names
      end if
    end repeat
    if names <> empty then put names into message box
    else put "no A grades" into message box
  end mouseUp
```

4. Click the Find A's button.

Result:

The Message box displays the following line:

```
Jason Blimp, Margo Dandiform, Narcissa Herculon, Alonzo Wunderbar
```

So four out of five students got an A on that quiz.

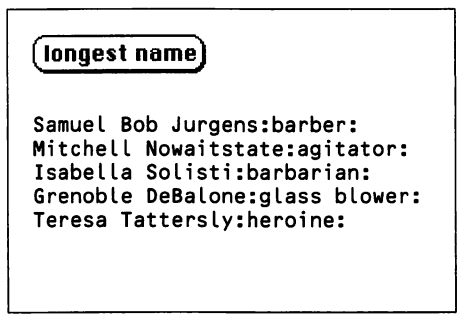


Figure 11-16. Name data.

In the script, we use the `number of` function to find out how many lines are in the field. Next, we initialize `names` to the `empty` string (a good practice). The `repeat with` loop then cycles through each line in the field. If the fifth item of the line contains A, the first item of the line (the name) is appended to the variable `names`. If we simply appended the name to `names`, the list would look odd: `Jason BlimpMargo DandiformNarcissa HerculonAlonzo Wunderbar`. Instead, we concatenate the name with the string `", "` to separate the words. However, we don't want a comma after the final name, so after the first name is put into `names`, we add the `", "` to the beginning of subsequent additions. That is, the second string added is `", Margo Dandiform"`. This makes the list more readable.

The *offset()* Function

The `contains` and `is in` operators tell whether one string is contained within another. The contained string is called a *substring*. The `offset()` function tells exactly where in a string the substring is found. Figure 11-17 shows its syntax.

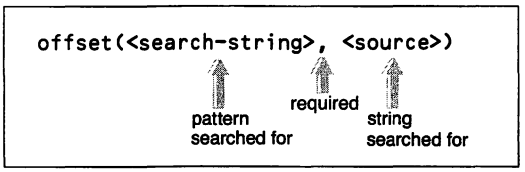


Figure 11-17. The `offset()` function syntax.

Note that the first argument (`<search-string>`) is the sought-for substring, and the second argument (`<source>`) is the string being searched. If `<search-string>` is not found, `offset()` returns the value `0`; otherwise it returns the character position, measured from the beginning of `<source>`, of the first chararacter of `<search-string>`. For instance, `offset("to", "got to run")` returns 5, because the `t` of `to` is the fifth character in the string `"got to run"`. Figure 11-18 shows the result of

```
offset("13th", "Friday the 13th") = 13
```

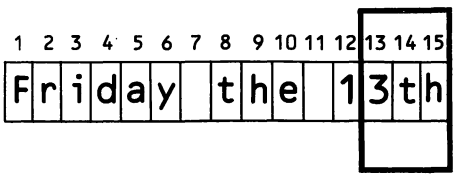


Figure 11-18. The `offset` function.

Let's take our traditional trip to the Message box to investigate the properties of this function.

Example 18:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Enter

   ```
   offset("go", "Go Go")
   ```

Result:

The Message box displays `1`, the starting location of Go in Go Go.

As usual, HyperTalk does not distinguish between upper- and lowercase. Also, `offset()` finds the first occurrence of the search string.

3. Enter

   ```
   offset(space, "Spring Ahead")
   ```

Result:

The Message box displays `7`.

Recall that `space` is the HyperTalk constant representing the space character. Of course, the real power of `offset()` is that the strings can be containers and

expressions as well as simple literal strings. Look at an example. Suppose that you have a little database consisting of a field in which each line contains some information about friends. Moreover, the field uses a colon to separate different kinds of information. Figure 11-16 shows an abbreviated example. Now suppose that you want HyperTalk to find the longest name in the field. You can't use `item` to isolate a name because the field uses colons rather than commas as separators. You can, however, use `offset()` to locate the first colon in each line. The larger the offset to the first colon, the longer the name.

Example 19:

1. Go to your HyperTalk Bible stack and the Strings card.
2. Create a field called `pals` and fill it with the information shown in Figure 11-16. Size the box so that each entry fits on one line.
3. Create a button called `Longest Name` and give it this script:

```
on mouseUp -- Longest Name button
  put the number of lines in card field "Pals" into nlines
  put 0 into max
  repeat with n = 1 to nlines
    put offset(":", line n of card field "Pals") into inset
    if inset > max then
      put inset into max
      put n into linenumber
    end if
  end repeat
  if max > 0 then
    put empty into longname
    repeat with n = 1 to max - 1
      put char n of line linenumber of card field "Pals"¬
      after longname
    end repeat
    put "The longest name is" && longname into the message box
  else
    put "No names found" into the message box
  end if
end mouseUp
```

4. Click the Longest Name button.

Result:

The Message box displays `Mitchell Nowaitstate`.

If two or more names have the same length, the script only reports the first of the tied names. If you like, you can modify the script to keep track of ties. Our purpose here is to illustrate several string operations. The first `repeat with` loop examines each line in the field in turn, finds the location of the first colon, and saves it in the variable `inset`. It compares this value (`inset`) to the value of `max`, which initially is zero. Whenever `inset` is larger than the current `max`, `max` is reset to the larger value. Also, the line number for the current longest name is saved in the variable `linenumber`.

The second `repeat with` loop uses the known length of the name (`max - 1`) and

the known line number (`linenumber`) to reconstruct the name character by character, thus extracting it from the whole line. Finally, we use string concatenation to report the result.

If you knew for certain that all the names were just two words, you could use the command `put word 1 to 2 of line linenumber of card field "Pals" into longname` rather than the loop. Then, however, you would be ignoring the possibility of names such as Geronimonoabolla and Peggy Sue Greenmulch.

The Selection

Macintosh programs characteristically let you select text by using the mouse or keystrokes. For example, in HyperCard, the Tab key selects text in text fields. Double-clicking the mouse on a word selects the word. Clicking and dragging the mouse through text selects that part of the text. Once text is selected, it appears in highlighted form, as shown in Figure 11-19.

```
Mighty-thewed Haxar
thumped the table with his
massive fist, yelled for
more ale, and began singing
"Haxar the Wonderous."
```

Figure 11-19. Selected text.

In HyperCard, `the selection` (or `selection`, for short) is a special container that represents the selected text. This allows you to create scripts that use selected text. Let's see how that works.

Example 20:

1. Go to your HyperTalk Bible stack and the Strings card.
2. Create a button called `Selection` and put this script into it:

```
on mouseUp -- Selection button
  if the selection is empty then
    put "No current selection" into the message box
  else
    put the selection into the message box
  end if
end mouseUp
```

3. Select some text from one of the fields, and then click the Selection button.

Result:

The selected text (or as much as fits) appears in the Message box.

The `selection` container truly is the selected text, not just a copy of it. Any changes you make to `selection` are reflected in the selected text on screen.

Example 21:

1. Go to your HyperTalk Bible stack and the Strings card.
2. Create a new button called `Tceles` and give it this script:

```
on mouseUp -- Tceles button
  if selection is empty then
    answer "Select some text first."
    exit mouseUp
  end if
  put empty into reverse
  repeat with n = length(selection) down to 1
    put char n of selection after reverse
  end repeat
  put reverse into selection
end mouseUp
```

3. Select some text and click the Tceles button.

Result:

The selected text is replaced by the same text, but in reverse order.

If you don't like the text in reversed form, select and reverse it again.

The `length` function tells how many characters are in `the selection`. Then we use a `down to repeat` loop to place the characters in reverse order into a variable called `reverse`. Finally, `reverse` is put into `selection`. When `selection` is used as a target, the original selected text is replaced by whatever we place in the container `selection`.

Selection Functions

Version 1.2 of HyperTalk added several new functions for dealing with selected text. One set of functions provides more detailed information about selections. Then, there is a command for simulating the selection of text from a script. We look at these new features in turn.

Selection Information

The new information functions are `the selectedText`, `the selectedField`, `the selectedLine`, and `the selectedChunk`. Figure 11-20 shows the syntax for these functions.

`the selectedText` function returns the text that has been selected. Thus, it has the same value as `the selection`. However, because it is a function, not a container, you can't put anything into `the selectedText`. `the selectedField` function identifies the container holding the selected text: Message box, for example, or card field 3.

```
the selectedText
selectedText()

the selectedField
selectedField()

the selectedLine
selectedLine()

the selectedChunk
selectedChunk()
```

Figure 11-20. The selection functions.

`the selectedLine` function identifies the line containing the selection, as in line 5 of card field 3. Finally, `the selectedChunk` function returns the character range of the selection, as in char 20 to 56 of card field 2.

Try these functions.

Example 22:

1. Go to your HyperTalk Bible stack and the Strings card.
2. Check that the Friends field still reads as follows:

```
Fifi LaBozena
Lefty Dorkus
Treakle, Nertley
Truhart, Tess
"Big Feet" Ben
```

3. Create a new button called `Select Info` and give it this script:

```
on mouseUp -- Select Info button
  put the selectedText & return & the selectedField &¬
  return & the selectedLine & return & the selectedChunk¬
  into card field "Samples"
end mouseUp
```

4. Use the mouse to select the name `Tess` in Friends, and click the Select Info button.

Result:

The following information appears in the Samples field:

```
Tess
card field 1
line 4 of card field 1
char 54 to 57 of card field 1
```

If the chunk contains more than one line, `the selectedLine` returns the number of the first line containing the selected text.

These new functions simplify the task of writing field scripts that react to the parts of the field that are selected.

The *select* Command

The `select` command lets you select text from a script. The syntax is shown in Figure 11-21.

```
select [before|after] <chunk> of <field>
select [before|after] <chunk> of msg box
select [before|after] text of <field>
select empty
```

Figure 11-21. The `select` command syntax.

The `before` and `after` options put the text insertion point at the specified location. If they are omitted, the specified text is selected. For example, `select after word 1 of card field 1` puts the text insertion point immediately after the first word of the field, but `select word 1 of card field 1` selects (and highlights) the first word itself. In general, `<chunk>` can be any of the chunk expressions mentioned earlier: character ⟨n⟩ of, word ⟨n⟩ of, line ⟨n⟩ of, item ⟨n⟩ of, and combinations and ranges of these. To refer to an entire field, use the `select text of <field>` form.

(You may wonder why you can't simply use `select <field>` to select the entire text. The reason is that the `select <field>` command simulates choosing the field tool and clicking a field. That is, it selects the field, not its text contents.)

Here's a simple example showing how this command works. Because the Strings card is getting full, let's open a new card.

Example 23:

1. Go to your HyperTalk Bible stack and create a new card called `Strings 2`.
2. Create a new field and place the following text into it:

```
If you can read this, you must
be a great scholar!
```

3. Open the field script and insert the following text:

```
on mouseEnter
  repeat with n = 1 to number of words in me
    select word n of me
    play "harpsichord" tempo 500 85 - length(word n of me)
    wait 20 ticks
  end repeat
  select empty
end mouseEnter
```

4. Select the browse tool and move the mouse into the new field.

Result:

Each word in the field is highlighted in turn, accompanied by a note keyed to the word length.

Note that the `select empty` command causes nothing to be selected. If we omit this command from the script, the last word remains selected when the script ends.

We used the `mouseEnter` message rather than a mouse button message because button messages are sent only to locked fields, whereas the `mouseEnter` message goes to locked and unlocked fields.

When text is locked, you can't use the mouse or keyboard directly to select text in the field (see chapter 6, "Messages"). However, you can use the `select` command to select text even in locked fields. Therefore, the last example still works if you lock the field.

Summary Script

You can use `contains`, `is in`, and `offset()` to search in places that are *terra incognito* for the `find` command. You can look through card names, scripts, and, as we show next, button names. Let's see how to search through a stack for all button names containing a given string. This can be useful if you are developing a stack with lots of buttons.

Example 24:

1. Go to the Strings card in your HyperTalk Bible stack.
2. Create a button called `Find Buttons` and give it the following script:

```
on mouseUp -- Find Buttons button
  ask "What button do you seek?"
  if it is empty then exit mouseUp
  put it into bname
  push card -- save original card

  -- first search for background buttons
  repeat with bkgrnd = 1 to the number of backgrounds
    go to bkgnd bkgrnd
    repeat with bkbutn = 1 to the number of bkgnd buttons
      put the short name of bkgnd button bkbutn into shortname
      if bname is in shortname then
        answer shortname & ": bkgnd" && bkgrnd &¬
        ", button #" && bkbutn with "continue" or "quit"
        if it is "quit" then exit mouseUp
      end if
    end repeat
  end repeat

  -- next, search for card buttons
  repeat with cardnum = 1 to the number of cards
    go to card cardnum
    repeat with butn = 1 to the number of card buttons
      put the short name of card button butn into shortname
      if bname is in shortname then
        answer shortname & ": card" && cardnum &¬
        ", button #" && butn with "continue" or "quit"
```

continued

```
        if it is "quit" then exit mouseUp
      end if
    end repeat
  end repeat

  pop card -- return to original card
  answer "Done!"
end mouseUp
```

3. Click the Find Buttons button.

Result:

A dialog box asks for your search string (see Figure 11-22).

4. Enter word.

Result:

The script finds buttons whose name contains the string word, such as the Word Counter button of Example 15. For each `find` you can quit, remaining on the card or background holding the button, or you can continue the search.

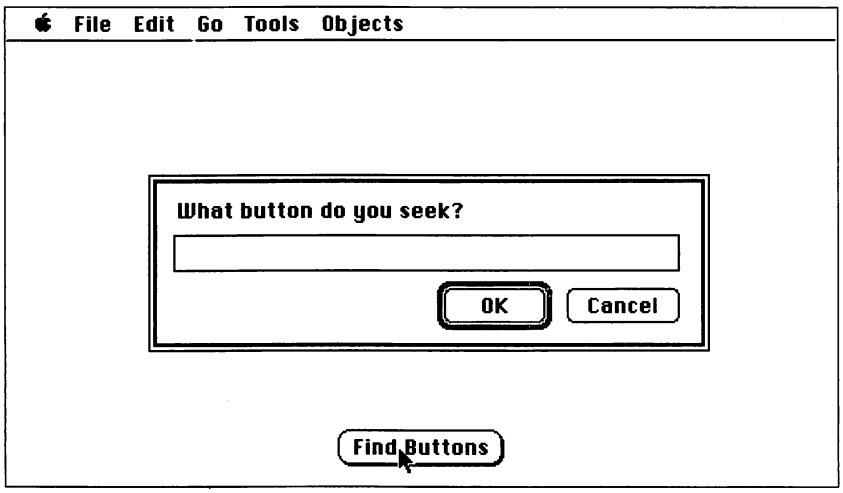


Figure 11-22. The Find Buttons dialog box.

This script is pretty long, so let's summarize its approach. The script must solve two problems. First, it must systematically locate all the background and stack buttons in a stack. Second, it must compare each button name with the search string to see whether there is a match. We can solve the first problem by using two `repeat for` loops. One loop searches through all backgrounds in the stack for background buttons, and the second searches through all cards for card buttons. We can solve the second problem—searching for a particular string—by using the `is in` operator. Now, look at a few details.

The script begins by requesting the user to enter a search string. If the user

fails to do so, the script terminates. Otherwise it saves the search string in the variable `bname` and pops the current card.

Next, we use the number of backgrounds to determine how many backgrounds the stack contains. The outer `repeat with` loop then cycles through the backgrounds. Within the loop, we use the `number of bkgnd buttons` to determine the number of buttons in a particular background. Then, we use an inner `repeat with` loop to check each button name for the search string. Note that we use `is in` rather than `is` when searching for the string because one doesn't always remember the precise name of a button. When you do a string comparison with `is` (or =), as described in chapter 9, "Ifs," you need to match the name exactly. By using `is in`, we can use, for example, count to find count or counts or counter. Also, if a family of buttons have parts of a name in common, such as travel, travel 1, and travel 2, we can find them all in the same search.

If there is a match, the script tells us so in an answer box, as shown in Figure 11-23. Note how concatenation is used to construct the message. At this point, users can continue or quit. If they quit, the `exit mouseUp` command terminates the script.

Next, the script investigates the card buttons. The method is identical to that for the background buttons, but this time the script cycles through cards rather than backgrounds.

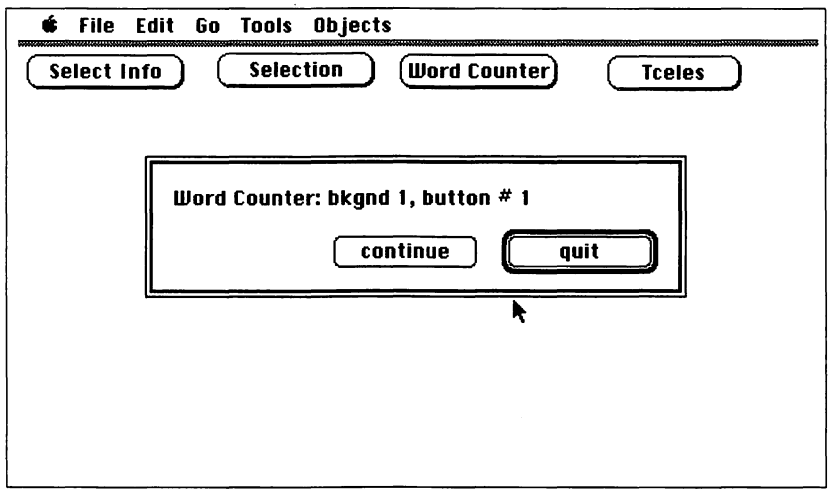


Figure 11-23. Find Button finds a button.

Finally, if the user hasn't quit, the script returns to the original card and uses an answer box to tell us it's done.

If you are ambitious, you can modify the script so that it prompts you to identify which stack to search. If you are even more ambitious, you can have the script prompt you for the next stack to search.

Summary

HyperCard supports many string activities. The `&` and `&&` operators glue strings together. The quotes construct literal strings like `"I'm a HyperTalk critter."` HyperTalk strings constants such as `space`, `empty`, and `return` facilitate the construction of strings. The `charToNum()` and `numToChar()` functions let you convert characters to ASCII codes and back. The `length()` and `the number of` functions give you the length of strings in characters, words, items, and lines. The `is in` and `contains` operators and the `offset()` function find whether a string contains a given substring. The `selection` container, the `select` command, and the family of selection functions provide handy tools for dealing with selected text. With these commands, functions, constants, and operators, you can perform a variety of string operations, including constructing sentences from different parts and performing searches.

Review Questions

1. What's the difference between `&` and `&&`?
2. Suppose that the variable `tree` is set to `"oak"`. Interpret the following concatenation:

   ```
   quote & "S" & tree & " the" && tree & "," & quote && "she said."
   ```

3. Which of the following can be used as a text source for a `put` command?

   a. `char 5 of card field 3`
   b. `line 3 of card 2`
   c. `word 4 of the message box`
   d. `line 4 to line 6 of card field 1`
   e. `the selection`
   f. `the selectedText`

4. Which choices in Question 3 can be used as destinations for a `put` command?
5. Give two HyperTalk commands to put the number of characters into card field 1 in the `fieldCount` variable.
6. Suppose that you want to use the Message box to append the string contained in the variable `newstuff` to card field Big Things and that you want the new material to start on a new line. What command do you type?
7. How do you specify in what units `the number of` function is to count?
8. Provide three ways to make the Mac beep if the variable `possibilities` contains the string `beep`.
9. Write a command that makes the Mac beep if the first character of the global variable `it` is a capital letter.

Exercises

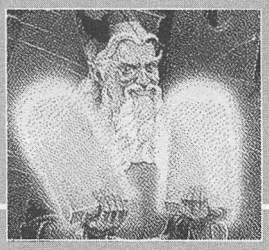
1. Write a script that finds the average number of characters per word in a field. For your first attempt, let the character count include all the spaces and punctuation.
2. Improve Exercise 1 by excluding spaces from the character count.
3. Write a field script so that when you click the field (choose Locked Text in the field dialog box to let you click the field), the script lists all the buttons in the stack in the following format:

   4 Background Button(s)
   To Do List, background 1, button 1
   Field Strings, background 1, button 2
   Word Counter, background 1, button 3
   Cat Box, background 2, button 1

   10 Card Button(s)
   Flowers, card 1, button 1
   exercise 2, card 1, button 2
   exercise 1, card 1, button 3
   longest name, card 4, button 1
   Cattle, card 5, button 1
   Find Buttons, card 5, button 2
   List Buttons, card 5, button 3
   ASCII Values, card 8, button 1
   concatenate, card 9, button 1
   Find A's, card 13, button 1

Chapter 12

Date and Time



Quick View

Commands
convert

Keywords
short date
long date
abbreviated date, abbrev date
dateItems
short time
long time

Functions
the date

Functions *cont'd*
the time
the seconds
the ticks

Concepts
Date and time formats
Using the date
European date and time
Getting the time of day
Combining the date and time
Date conversions
Measuring elapsed time

Chapter 12

Date and Time

Through the ages, much has been said about time. "Time and tide wait for no man." "Time, time, time, is on my side. Yes, it is." "Time is money." HyperTalk is more practical in its approach; it just reports the facts. In particular, it can tell you the time, the date, and elapsed times. It can convert time and date formats for you. You can do a lot with these tools: Keep track of when you use a stack, calculate time and date differences, time a script, put a display "clock" on a card, and so on. This chapter presents HyperTalk's date and time functions and commands and shows you how to use them. You'll find the date and time commands useful for any stack involving scheduling, appointments, record keeping, calendar functions, elapsed time in tasks or games, and so on.

the date Function

The basic HyperTalk tool for finding the date is a function appropriately named `the date`. This function returns the current date in the format month/day/year, but by using modifiers, you can get other formats. The syntax for `the date` is shown in Figure 12-1.

```
the date
date()
the short date
the long date
the abbreviated | abbrev | abbr date
```

Figure 12-1. The `the date` function syntax.

You can use `the date` or `date()` interchangeably; either `the` or `()` indicate that `date` is a function. But to use `the date` modifiers, you have to use the `the` form. Table 12-1 shows the formats HyperTalk provides.

Table 12-1. Date Formats

Function	Format	Example
the date	month/day/year	4/14/89
date( )	month/day/year	4/14/89
the short date	month/day/year	4/14/89
the long date	dayname, month day, year	Friday, April 14,1989
the abbreviated date	dayAbbr, mon day, year	Fri, Apr 14, 1989
the abbrev date	dayAbbr, mon day, year	Fri, Apr 14, 1989
the abbr date	dayAbbr, mon day, year	Fri, Apr 14, 1989

Note that `the date`, `date()`, and `the short date` produce identical formats. The `abbreviated date` function gives a more compact representation of the date. Because the command name is so long, HyperTalk provides two equivalent abbreviations for the `abbreviated date`: `the abbrev date` and `the abbr date`.

`the date` function is simple to use. If you like, you can go to any stack, open the Message box, and try these various forms.

Using *the date* Function

It probably won't take many visits to the Message box to get bored with that use of `the date`, so let's find a more interesting application.

Example 1:

1. Go to your HyperTalk Bible stack and create a new card called `Date and Time`.
2. On this card create a button called `Today's Date` and give it this script:

```
on mouseDown -- Today's Date button
  set the name of me to the date
end mouseDown


on mouseUp
  set the name of me to "Today's Date"
end mouseUp
```

3. Position the cursor over Today's Date, hold the mouse button down for a second or so, and then release it.

Result:

While the mouse button is down, the current date appears in the button box. When you release the mouse button, the button name reappears.

Here the phrase `of me` represents the button to which the message being handled was sent, so the `mouseDown` message causes the button to be renamed to the current date. Then, the `mouseUp` message acts to reset the name to the original value.

Suppose that you just want part of the date. You can use HyperTalk's text chunk specifiers to access components of the date.

Example 2:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Enter

```
word 1 of the date
```

Result:

The Message box displays the date in this format: `9/30/89`.

HyperTalk does not recognize the / as a word separator, so `9/30/89` counts as a single word. In HyperTalk, only spaces, tabs, and returns separate words. Similarly, `item 1 of the date` also is `9/30/89`, because only commas indicate separate items.

3. Enter

```
word 1 of the long date
```

Result:

The Message box displays the day of the week in this format: `Saturday,`.

Because there is no space between the `y` in `Saturday` and the comma, the comma counts as part of the first word.

4. Enter

```
item 1 of the long date
```

Result:

The Message box displays the day of week in this format: `Saturday`.

Items are comma-separated, so the "item" approach is the best one.

5. Enter

```
item 2 of the long date
```

Result:

The Message box displays month and day in this format: `September 30`.

Item number 2 consists of two words, because there isn't a comma between the month and the day.

6. Enter

```
word 1 of item 2 of the long date.
```

Result:

The Message box displays just the month name.

By using `word` and `item`, you can access any part of the `long date` value.

There is one tricky aspect about using `item` with `the long date`. See whether you can notice it in the next example:

Example 3:

1. Go to the Date and Time card in your HyperTalk Bible stack and open its script.
2. Enter the following text there; for testing purposes, use the current date rather than "April 1."

```
on openCard
  if item 2 of the long date is not "April 1" then
    beep 3
    answer "IRREMEDIABLE SYSTEM FAILURE: GOODBYE!" with "OK"
  end if
end openCard
```

3. Leave the card and then return to it.

Result:

An answer box displays the disheartening message in Figure 12-2.

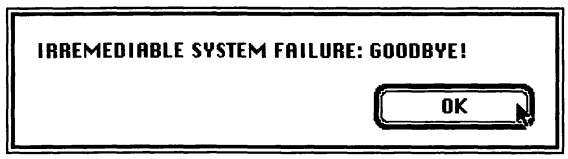


Figure 12-2. Ominous answer box.

The trick is that in the `if` comparison, you need to precede the month with a space. That's because the format for `the long date` (Friday, April 1, 1988) has a space after each comma. That space counts as part of the item. If you replace your equivalent of "April 1" with the equivalent of " April 1" (note the leading space), you don't get the terrifying warning of Figure 12-2. (Incidentally, you probably should remove this example when you're done. Otherwise, you may open this card a month or so later, get the answer box message, and forget it was just an example.) If you don't want the space in the comparison, you can try `word 1 to 2 of item 2 of the long date`.

Tiny Pictures

If you like the little pictures of cards that HyperCard shows when you Go Recent (Command-R), you can put these pictures into your stack! To get a card represented in this tiny format, go to the card and do a Copy Card. Then, go to the card into which you want to paste the tiny picture. Now, do a Paste with the Shift key down. A tiny card appears.

European Date and Time

Europeans use a different format for the date. For example, they show July 4, 1776, as 4 July 1776. European versions of HyperCard conform to that format. Either form is easily interpreted when the month is spelled out, but you better know which form you're reading when you see the short forms (7/4/76 versus 4/7/76)!

Incidentally, you can use HyperTalk text chunks to convert U.S.-style dates to European-style dates, as the next example shows.

Example 4:

1. Go to the Date and Time card of your HyperTalk Bible stack.
2. Clone the Today's Date button and open the new button.
3. Rename the button `Today's Eurodate`.
4. Modify the button script to read as follows:

```
on mouseDown -- Today's Eurodate button
  get the long date
  set the name of me to word 2 of item 2 of it && word 1 of item
  2 of it & item 3 of it
end mouseDown

on mouseUp
  set the name of me to "Today's EuroDate"
end mouseUp
```

5. Move the cursor to Today's Eurodate, hold down the mouse button briefly, and then release it.

Result:

While the mouse button is down, the button displays the current date European style: 15 April 1988.

Because all this button does is switch the first and second words of the date, it also would convert a European-style date to the U.S. style, so if you have a European version of HyperCard, you can use this same example, but change the button name to, perhaps, Ameridate.

Because the 4/15/88 form is a single word and item, you can't easily divide it into its component parts to construct the European 15/4/88. However, HyperTalk provides one more format (`dateItems`) that gives you access to each part. You see that form when we discuss the `convert` command. Because `dateItems` handles times as well as dates, we discuss the time function first.

Getting *the time* of Day from Your Macintosh

The HyperTalk function for obtaining the time is `the time`. It has the syntax shown in Figure 12-3.

```
the time
the short time
the long time
```

Figure 12-3. The `the time` function syntax.

The four forms of the command provide two time formats, as shown in Table 12-2.

Table 12-2. Time Formats

Function Example	Format	Example 12-hour Clock	Example 24-hour Clock
the time	hours:minutes (AM/PM)	10:23 AM	10:23
time( )	hours:minutes (AM/PM)	7:42	19:42
the short time	hours:minutes (AM/PM)	12:09 AM	0:09
the long time	hours:minutes:seconds (AM/PM)	8:31:47 PM	20:31:47

Whether you get the 12-hour clock or the 24-hour clock depends on how the Control Panel for your Macintosh is set.

There is no real mystery to using `the time` function, as a few seconds of Message box work shows you. As with the output of `the date` or of any other function, you can display the date returned by `the time` function, you can assign it to a container, or you can incorporate it into an expression or string. Here, for instance, is a display example borrowed from the Home stack.

Example 5:

1. Go to the Date and Time card of your HyperTalk Bible stack and create a field called `Time`, checking the Lock Text box.
2. Open the card script and insert the following text:

```
on idle
  put the time into card field "Time"
  pass idle
end idle
```

3. Return to the card.

Result:

The current time appears in the field.

The time is updated each `idle` message, about once every twelfth of a second as long as the mouse button is up. The `pass idle` command sends the `idle` message on so that it reaches any idle handler higher in the heirarchy. You can select the font, style, and size you like for the time display and reposition and resize the field to fit your needs. See Figure 12-4.

11:35 AM

Figure 12-4. A time field.

Now, let's incorporate the time into a longer string. We haven't done anything with the `openField` message for a while, so let's make it part of the example.

Example 6:

1. Go to the Date and Time card of your HyperTalk Bible stack.
2. Create a new field, position and size it to your liking, and give it the following script:

```
on openField
  answer "It's" && the time & ". Just thought you should know."¬
  with "OK"
end openField
```

3. Move the browse tool into the new field and click it to select the text cursor.

Result:

You get a message similar to that shown in Figure 12-5.

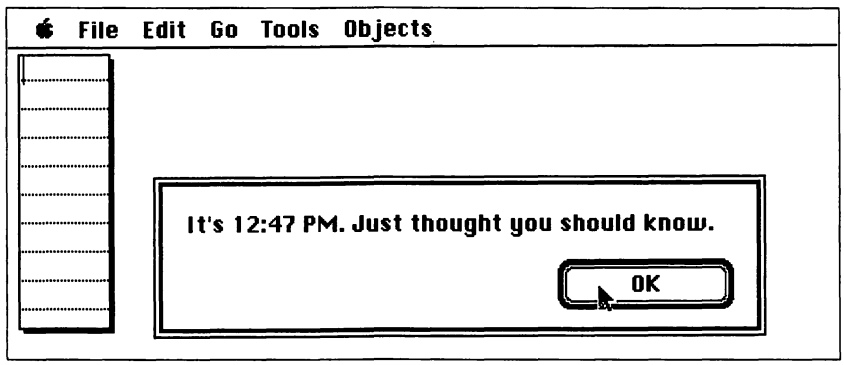


Figure 12-5. A timely reminder.

The main point here is that you can treat the output of the time like any other HyperTalk value.

Combining Date and Time Information

Because the output of `the date` and `time` functions are strings, you can join them into a larger string. Here's an example that combines output from the two functions in a script that maintains a log of when you use a stack.

Example 7:

1. Go to your HyperTalk Bible stack and create a card called `Log Card`.
2. In that card, create a card field called `Log Field`, make the field a scrolling one, and choose a smaller font such as Monaco 9.
3. Place the following text into the stack script:

```
on openStack
  put "Stack opened:" && the time & "," && the abbrev date &¬
```

continued

```
  return after card field "Log Field" of card "Log Card"
end openStack

on closeStack
  put "Stack closed:" && the time & "," && the abbrev date &¬
  return after card field "Log Field" of card "Log Card"
end closeStack
```

4. Go to your Home Stack, and then return to your HyperTalk Bible stack.

Result:

The card field shows the time and date that you left the stack (the `closeStack` message) and that you returned (the `openStack` message).

Recall that the return constant serves to begin a new line and, therefore, to put each log entry on its own line.

Date Conversions

HyperTalk's `convert` command provides additional date formats. Normally, you apply this command to a container (such as a variable or a field) containing the date and specify the desired format. The command then converts the date as instructed and places the result in the original container. Figure 12-6 shows the syntax for this command.

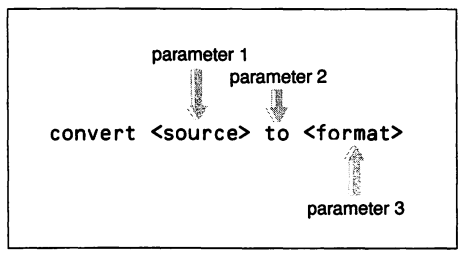


Figure 12-6. The `convert` command syntax.

Table 12-3 shows the choices you have for `format`. Please note that the `dateItems` format uses the 24-hour clock for the hour, and the `day-of-week` takes Sunday to be day 1.

The `dateItems` format uses the 24-hour clock for the hour. The `day-of-week` takes Sunday to be day 1.

This list includes the formats supplied by `the date` and `the time` functions, the multi-item `dateItems` format, which supplies both date and time information, and the `seconds` format, which converts everything since 00:00, January 1, 1904 to seconds. The `seconds` format is very useful for finding the time between two dates.

Let's go to the Message box and see how the `convert` command works.

Table 12-3. Date Conversion Formats

Modifier	Format	Example
short date	month/day/year	6/8/88
long date	day-name, month day, year	Wednesday, June 8, 1988
abbreviated date	d-name, mon day, year	Wed, Jun 8, 1988
abbrev date	d-name, mon day, year	Wed, Jun 8, 1988
dateItems	year,month,day,hour,minute,second,day-of-week	1988,6,8,15,23,11,4
short time	hour:minute (AM/PM)	3:23 PM or 15:23
long time	hour:minute:second (AM/PM)	:23:11 PM or 14:23:11
seconds	seconds since 12 AM, 1/1/04	2664631391

Example 8:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Enter in the Message box

```
put "2/4/68" into tempura
```

3. Enter

```
convert tempura to abbrev date
```

4. Enter `tempura`.

Result:

The Message box shows `Sun, Feb 4, 1968`.

The `convert` command placed the converted date into the original container, `tempura`.

5. Enter in the Message box

```
convert tempura to seconds
```

6. Enter `tempura`.

Result:

The Message box shows `2022624000`.

This is the number of seconds elapsed from 12 AM, January 1, 1904, to 12 AM, February 4, 1968. Because the original date did not include time information, HyperTalk takes the hours, minutes, and seconds to be 0.

7. Enter in the Message box

```
convert tempura to dateItems
```

8. Enter `tempura`.

Result:

The Message box displays `1968,2,4,0,0,0,1`.

The `dateItems` format indicates that you are dealing with the year 1968, the month 2 (February), and day 4 of the month. The hour, minutes, and seconds items all are set to 0, and the terminating 1 indicates the first day of the week, Sunday.

9. Enter in the Message box

```
convert tempura to short date
```

10. Enter `tempura`.

Result:

The Message box displays `2/4/68`, the original form.

These entries demonstrate the normal method of using `convert`: applying it to a container and having it update the container. You can apply `convert` to a literal string or to the output of a command, too. In cases where you fail to provide a container, the result is placed in that familiar global variable, `it`.

Example 9:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Enter in the Message box

```
convert "1/1/90" to long date
```

3. Enter `it`.

Result:

The Message box displays `Monday, January 1, 1990`.

The string `"1/1/90"` isn't a container, so `convert` can't place the converted date there, but good old `it` is happy to oblige.

4. Enter in the Message box

```
convert the date to dateItems
```

5. Enter `it`.

Result:

The Message box displays information in the form `1988,4,12,0,0,0,3`.

This time the source for `convert` is a function (`the date`). Again, `convert`, lacking a container to store its output, uses the global variable `it`. All items are set to 0, because the function `the date` doesn't supply any time information to convert. Thus, the conversion really is to the beginning of the day at 00:00:00. The `time` function is more detailed.

6. Enter in the Message box

```
convert the time to dateItems
```

7. Enter `it`.

Result:

The Message box displays information in the form `1988,4,12,14,51,0,3`.

Here, both time and date information are present in a comma-separated list of seven items. The time, in this case, is to the nearest minute because that is what `the time` reports. By using `the long time`, you get the seconds also.

The `seconds` and `dateItems` formats are useful for making date and time calculations, because they present information in numerical form. For instance, adding 1 to the second item of a `dateItems` format updates the stored date by a month. You can use `seconds` to find time differences.

Example 10:

1. Go the Date and Time card of your HyperTalk Bible stack.
2. Create a button called `Your Age` and put this script into it:

```
on mouseUp -- Your Age button
  ask "Enter your birthdate in the format shown" with "1/24/58"
  if it is empty then exit mouseUp
  put it into birthtime
  convert birthtime to seconds
  convert the date to seconds -- result goes into "it"
  put it - birthtime into secondsOld -- approximate age in seconds
  put secondsOld / 86400 into daysOld -- 86400 seconds in a day
  answer "You are" && daysOld && "days old. Enjoy!"
end mouseUp
```

3. Click the Your Age button.

Result:

The dialog box shown in Figure 12-7 appears.

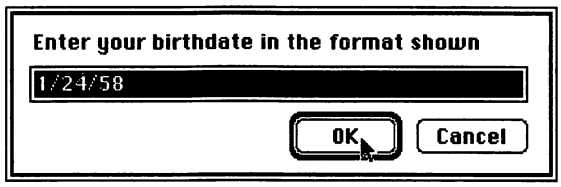


Figure 12-7. The Your Age dialog box.

4. Click OK to accept the default date.

Result:

An answer box like that shown in Figure 12-8 appears.

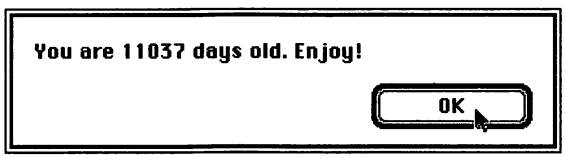


Figure 12-8. Your Age response.

Both the user input and the value returned by `the date` give the time to the beginning of the day (00:00 hrs), so the variable `secondsOld` corresponds to a whole number of days.

(If you want your true age in seconds, you have to input the exact time of your birth and use `the long time` rather than `the date` to get the exact current time. If accuracy is essential, you have to allow for daylight savings. This complicates things, because at different times and places, daylight savings has been implemented for

different intervals or not at all, so we just put this paragraph in parentheses and forget the matter.)

Finding a Stack's Modification Date

The `convert` function can be useful when you want to determine the date that a stack was last modified. By using the `version` function first, HyperCard 1.2 returns five comma-separated strings that contain a stack's creation and version information. The fifth and last string is the most recent modification date in the number of seconds since Jan 1, 1904. The following script uses `convert` to change the number of seconds item into a date in the format mm/dd/yy.

```
on mouseUp
   get the version of this stack
   convert item 5 of it to date
   put item 5 of it into msg
end mouseUp
```

Measuring Elapsed Time: *the seconds* and *the ticks* Functions

HyperTalk has two more time functions: `the seconds` and `the ticks`. As you probably can guess, `the seconds` measures time in seconds, whereas `the ticks` measures time in ticks, where one tick is 1/60th of a second. The syntax for `the seconds` function is shown in Figure 12-9.

```
the seconds
the secs
seconds()
secs()
```

Figure 12-9. The `the seconds` function syntax.

Figure 12-10 shows the syntax for `the ticks` function.

```
the ticks
ticks()
```

Figure 12-10. The `the ticks` function syntax.

Just as with `the seconds` format of convert, so also the function `the seconds` measures time from 00:00:00, January 1, 1904. This means that you can use `convert`

to convert the output of `the seconds` to another form. For instance, the command `convert the seconds to the date` puts the date in the form mm/dd/yy into the global variable `it`. The function `the ticks`, however, measures time in ticks from the last time the Macintosh was started. If your Macintosh has been on for two hours, `the ticks` returns the value 432000, because that is the number of ticks in two hours.

You can use these two functions to measure elapsed times. For example, consider the `repeat for` and the `repeat with` loops. The second form explicitly provides a loop variable that increments by one each loop. You can produce the same effect with the `repeat for` loop by providing and updating your own variable. Offhand, you'd expect that the do-it-yourself approach would run more slowly than the built-in loop, but does it? You can use `the seconds` to time each loop and learn the answer.

To emulate a `repeat with` loop with a `repeat for` loop, do this:

```
put 1 into n
repeat for 1000 times
  add 1 to n
end repeat
```

This imitates the following loop:

```
repeat with n = 1 to 1000
end repeat
```

Each has a variable `n` that varies from 1 to 1000. You could give the loops something to do, such as sum the integers from 1 through 1000, but that would obscure the time needed just for the loops themselves, so run empty loops:

Example 11:

1. Go to the Date and Time card of your HyperTalk Bible stack.
2. Create a button called `Test the Loops` and give it the following script:

```
on mouseUp -- Test the Loops button
  put 1 into n -- initialize to 1
  put the seconds into startTime
  repeat for 1000 times -- emulate "with" loop
    add 1 to n
  end repeat
  put the seconds - startTime into forTime

  put the seconds into startTime
  repeat with n = 1 to 1000
    -- empty loop
  end repeat
  put the seconds - startTime into withTime
  put "repeat for loop:" && forTime && "seconds; repeat with loop:"¬
  && withTime && "seconds" into message box
end mouseUp
```

3. Click the Test the Loops button.

Result:

On a standard Macintosh Plus using HyperTalk Version 1.2, the Message box shows the following message:

```
repeat for loop: 19 seconds; repeat with loop: 2 seconds
```

The two lines that time the first loop are these:

```
put the seconds into startTime
put the seconds - startTime into forTime -- find elapsed time
```

The first line gets and saves the time just before the loop starts. The second line gets the time just after the the loop ends, subtracts the starting time, and puts the result into the variable `forTime`.

For empty loops, at least, using the built-in counter of the `repeat with` loop is nearly ten times as fast as adding your own counter to a `repeat for` loop, so the obvious expectation is confirmed. If your Macintosh runs too fast for this test, you can increase the number of iterations in the two loops to make the loops run longer, or you can switch to `the ticks`, which measures time more precisely.

Rather than repeat the repeated repeat example, let's do something different to demonstrate `the ticks`. Here is a reaction-time example:

Example 12:

1. Go to the Date and Time card of your HyperTalk Bible stack.
2. Create a button called `This is a Test` and give it this script:

```
on mouseUp -- This is a Test button
  put "When the Mac beeps, quickly press the option key!" into
  the message box
  wait for 60 + random of 30 ticks
  beep
  put the ticks into count
  wait until the optionKey is down
  put the ticks - count into elapsed
  put "You took" && elapsed && "ticks, or" && elapsed/60¬
  && "seconds" into the message box
end mouseUp
```

3. Click the This is a Test button.

Result:

The Message box informs you, `When you hear the beep, press the option key.`

4. Follow those directions.

Result:

The Message box gives you a reaction-time measure, as shown in Figure 12-11.

The basic idea is simple: Note the time just after the Mac beeps and subtract it from the time just after the user presses the option key. The command `wait for 60 + random of 30 ticks` adds a bit of sophistication. It gives a variable time delay between

clicking the button and the beep; this makes it harder to anticipate exactly when the beep will sound.

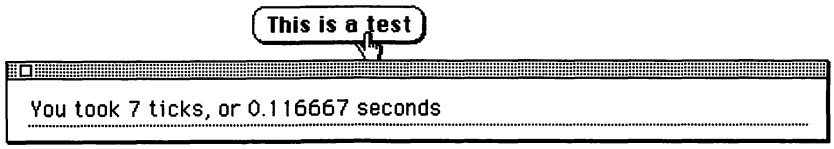


Figure 12-11. A reaction-time report.

Elapsed Time Conversions

You've seen how to measure elapsed times in ticks and in seconds. Suppose that you want elapsed time in minutes? You can, of course, divide the seconds by 60 to get a decimal value for minutes. That is, 90 seconds converts to 1.5 minutes. More typically, you want to express that as 1 minute, 30 seconds. You can do this conversion by using the `div` (integer division) and `mod` (modulus) operators. Consider the example of 90 seconds. The expression `90 div 60` evaluates to 1, because integer division discards any fractional parts. That gives the minute value. Next, `90 mod 60` yields 30, because the `mod` operator gives the remainder; that's the seconds value. We can extend this idea to convert the remaining minutes to hours and minutes and the remaining hours to days and hours. Here's a script that does just that:

Example 13:

1. Go to the Date and Time card in your HyperTalk Bible stack and open the Message box.
2. Enter in the Mesage box

```
put the seconds into cardSeconds
```

3. Now open the Date and Time card script and enter the following handlers. Note: Example 5 used an on openCard handler. You can replace it with the following on openCard handler, place the new handler above the old one (that causes the old one to be ignored), or add the command lines from the new handler to the old handler.

```
on openCard
  global cardSeconds
  put the seconds into cardSeconds -- save opening time
end openCard

on closeCard
  global cardSeconds
  put the seconds - cardSeconds into secondsUsed -- time used
  put secondsUsed div 60 into minutesUsed
```

continued

```
  put secondsUsed mod 60 into secondsUsed
  put minutesUsed div 60 into hoursUsed
  put minutesUsed mod 60 into minutesUsed
  put hoursUsed div 24 into daysUsed
  put hoursUsed mod 24 into hoursUsed
  answer "Card used" && daysUsed && "days," && hoursUsed &&¬
  "hrs," && minutesUsed && "mins," && secondsUsed && "secs"
end closeCard
```

4. Return to the card level, and then switch to another card.

Result:

An answer box reports the time you used.

The on openCard handler initializes the `cardSeconds` variable to the current time. Unfortunately, you already opened the card, so this step is skipped the first time you use this script. Therefore, you initialized `cardSeconds` manually in Step 2 just this once. The next time you return to the card, the on openCard handler does the job automatically.

The on closeCard handler determines how much time, in seconds, has elapsed since `cardSeconds` was initialized. Then, it uses the `div` and `mod` approach to convert the seconds figure to days, hours, minutes, and seconds.

Summary Script

A very common calendar question is, "On what day of the week does a particular date fall?" The HyperTalk date functions provide the tools to answer this question. Let's set up a button to do that:

Example 14:

1. Go to the Date and Time card of your HyperTalk Bible stack.
2. Create a button called `Any Day` and give it this script:

```
on mouseUp -- Any Day button
  ask "Enter date to be converted in mm/dd/yy form:" with the date
  put it into input
  convert it to long date
  put word 1 of it into dayOfWeek
  answer input && "is a" && dayOfWeek
end mouseUp
```

3. Click the Any Day button.
4. A dialog box prompts you to enter a date and offers the current date as a default; reply to the dialog box with 7/4/89.

Result:

The Message box displays `7/4/89 is a Thursday`.

This is a rather simple summary script, but the `date` and `time` functions are rather simple, too. The example does show several of the main points in this chapter.

First, you can use `the date` function to obtain the current date. Second, the return value of the `date` can be used like any other value; here it's used as a parameter for an `ask` command. Third, the `convert` command lets you convert from one date format to another. Fourth, you can use the HyperTalk text chunk expressions to access particular parts of a date value.

Review Questions

1. Which function calls give you the date in the following forms?

   a. 7/28/82
   b. Wednesday, March 18, 1987
   c. Fri, Nov 17, 1989

2. Consider our Today's Date button (Exercise 1). What if you hold the button down until the next day? Will the date on the button change?
3. We can get away with assigning the string `Beanie` to a variable by typing `put Beanie into nickname` as long as Beanie is not the name of a variable; HyperTalk understands that the quotes should be there. Can we similarly assign a short date to a variable this way?

   ```
   put 4/15/88 into taxtime
   ```

4. When you type the time on your Mac, you get 10:34 AM. Your friend, typing the time on her Mac, gets 10:34 without the AM. What's going on?
5. The variable `car` holds the date 7/22/82. Write commands to place the following in the variable result:

   a. the long version of 7/22/82
   b. the day of the week for 7/22/82
   c. the number of days from 7/22/82 to today

6. There are 60 ticks to a second. How, then, is `the seconds` different from `the ticks/60`?
7. How can you construct a sentence in the following form for the Message box? `It is 8:24 on a Sunday in July.` The date and time should be the current values.
8. How can you put the following sentence into the Message box? `The time is 14 hours, 32 minutes, and 10 seconds.` The time values should be the current ones, aside from the processing time needed to produce the message.

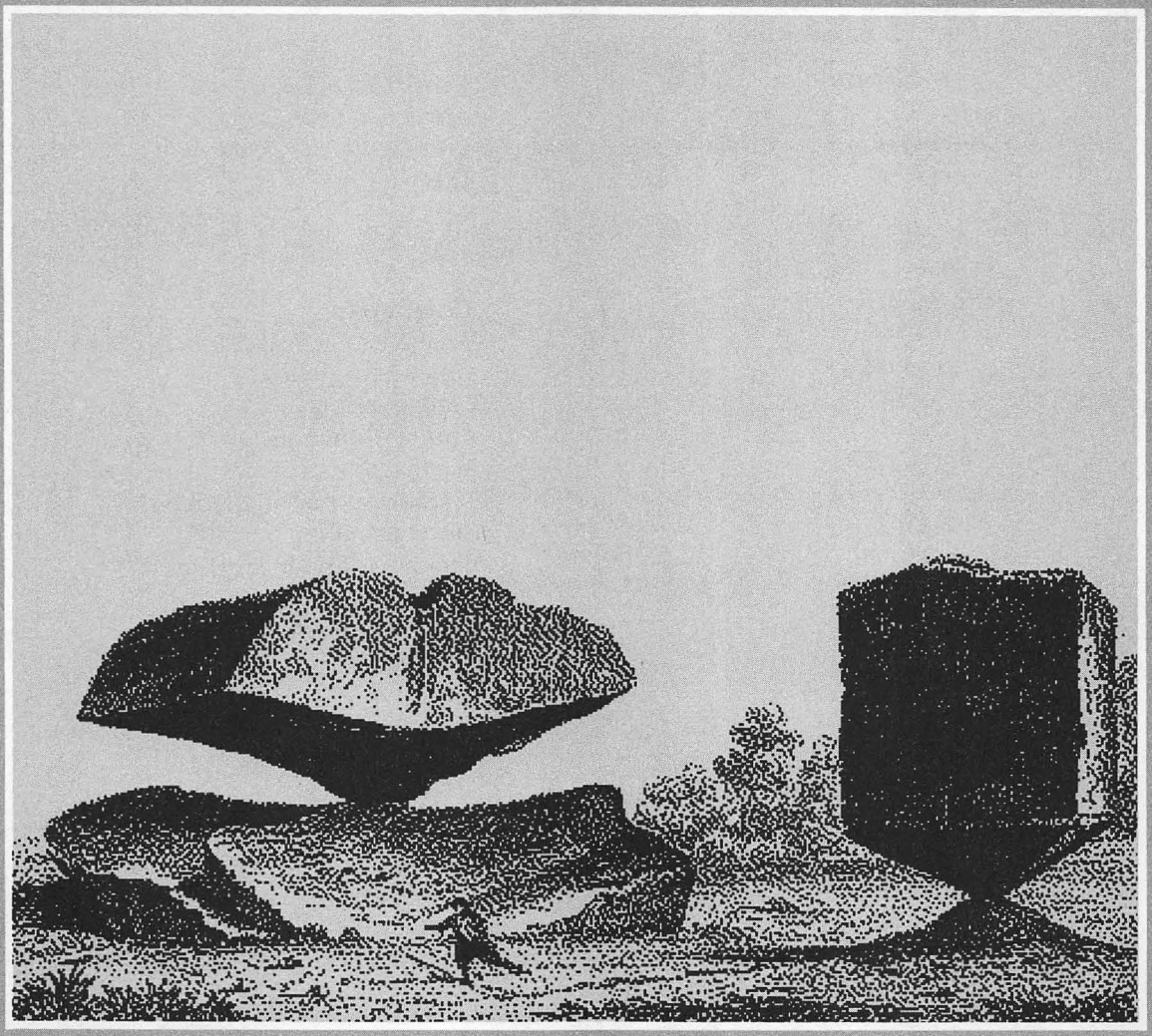
Exercises

1. Create a button called `Day of Week` that shows the day of the week when the mouse enters the button and reverts to displaying `Day of Week` when the mouse leaves the button.
2. Create a variation of the Today's Eurodate button that shows the date in the style day/month/year.

3. Create a button that reports in the Message box how long, in ticks, you held the mouse button down.
4. Modify the log script in the stack (Example 7) so that it puts into the log file a line saying how long, to the nearest whole minute, you had the stack open. This line should come between the lines saying when the stack was opened and when it was closed.

Chapter 13

Math Functions



Quick View

Functions

abs()
average()
max()
min()
round()
trunc()
random()
sqrt()
compound()
annuity()
sin()
cos()
tan()
atan()
exp()

Functions *cont'd*

exp1()
exp2()
ln()
ln1()
log2()

Concepts

Review of general math functions
Financial functions
Compounding
Annuity functions
Mortgages
Trigonometric functions
Circular motion
Exponents and logarithms

Chapter 13

Math Functions

In chapter 5, "Simple Math," we promised (threatened?) to give you more complete coverage of HyperTalk's math functions. You may wonder whether you need them. If you want to calculate compound interest, generate random movement, find points on a circle, or model population growth, HyperTalk's math functions are a big help. Even if you don't have a particular need in mind, you may get some ideas reading about these functions. As we talk about how these functions work, we also give examples of when you can use them.

In this chapter, we look at four groups of math functions: general functions, financial functions, trigonometric functions, and logarithmic and exponential functions.

General Math Functions

Because the general functions are those that don't fall into any one category, we won't try to categorize them other than to say they often come in handy. Many of them (`abs()`, `round()`, `trunc()`, `random()`, and `average()`), you've seen already in chapter 5, "Basic Math," but we mention them again for completeness.

Like other HyperTalk functions, these functions accept a list of parameters or arguments and return a value. Being math functions, they return a number. Those functions that just take one parameter can be expressed two ways, for example, `round(23.4)` and `the round of 23.4`. Those that take more parameters use parentheses only, with the parameters separated by commas, as in `average(88,92,90)`. A parameter can be a number, a container holding a number, a function or property returning a number, or an expression using any or all of the preceding.

Review: The *abs()*, *average()*, *round()*, and *trunc()* Functions

The `abs()` function returns the absolute value of its argument. (See Figure 13-1 for syntax.)

```
the abs of <number>
abs(<number>)
```

Figure 13-1. The `abs()` function syntax.

The absolute value means the magnitude of the number shorn of its sign. Thus, `abs(6)` and `abs(-6)` both are 6.

The `average()` function takes a list of comma-separated numeric values, as shown in Figure 13-2.

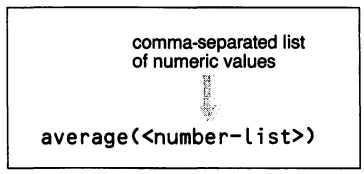


Figure 13-2. The `average()` function syntax.

This function returns the average value of its parameters. It's okay to have a trailing comma in the list. So `average(3,4,5,)` is equivalent to `average(3,4,5)`. This means you can build a list in, say, a HyperTalk loop by adding new numbers in the form `n`, to the existing list.

The `round()` and `trunc()` functions convert numbers containing fractional parts, such as 2.3 and π, to whole numbers. The syntax is simple, as shown in Figure 13-3.

```
the round of <number>
round(<number>)

the trunc of <number>
trunc(<number>)
```

Figure 13-3. The `round()` and `trunc()` syntax.

The difference between the two, recall, is that `round()` rounds to the nearest whole number, while `trunc()` always rounds down. That is, it truncates the fractional part, leaving the largest whole number contained in the original value. So `trunc(1.998)` is 1. Also, `round()` rounds numbers exactly halfway between whole numbers, such as 3.5 and 4.5, to the nearest even integer:

One other question is how negative numbers are treated. Normally, -2 is considered to be smaller than -1. Does that mean `trunc(-1.3)` is -2? Or does `trunc()` work with the absolute values and truncate -1.3 to -1? The answer is that `trunc()` truncates towards zero, so `trunc(-1.3)` is -1.

Let's look at a couple of useful tricks with `trunc()` and `round()`. First, you can use `trunc()` to obtain the fractional part of a number.

Example 1:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Enter in the Message box

```
pi - trunc(pi)
```

Result:

The Message box displays `0.141593`.

Here pi is the HyperTalk constant representing π. Subtracting the largest whole number in a number from the number itself leaves the fractional part.

Second, suppose you want to round a number to the nearest thousand rather than to the nearest integer. Then, you can divide the number by a thousand, round that, and then multiply the answer by a thousand:

3. Enter in the Message box

```
round(245118/1000) * 1000
```

Result:

The Message box displays `245000`.

Dividing 245118 by 1000 gives 245.118. Rounding gives 245, and multiplying by 1000 yields 245000.

The *max()* and *min()* Functions

The `max()` and `min()` functions take a list of parameters and return the largest and smallest values in the list respectively. Their syntax is shown in Figure 13-4.

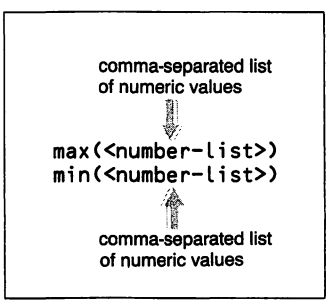


Figure 13-4. The `max()` and `min()` functions syntax.

The parameter list is a comma-separated list of numeric values and can contain one or more numbers. So `max(3)` is `3`, `max(12,12)` is `12`, and `max(-8,-2,-5)` is `-2`. The last examples demonstrate that negative numbers are considered to be smaller than positive numbers and that the more negative a number is, the smaller it is.

The min() and max() functions take the same form of parameter list as average(), so all three can be applied to the same data. Let's look at an example that uses these three functions plus abs() to analyze some rainfall data, say, nine years of data for April showers.

Example 2:

1. Go to your HyperTalk Bible stack and create a card called Math Card.
2. Create a new field called Rainfall and put these numbers into the field:

   10,15,20,12,18,14,13,16,17

3. Create a new button called Rain Data and give it the following script:

```
on mouseUp -- Rain Data button
  put the number of items in card field "RainFall" into num
  put 0 into sum
  put average(card field "RainFall") into ave -- compute average
  repeat with n = 1 to num
    add abs(ave - item n of card field "RainFall") to sum
  end repeat
  set the numberFormat to "0.00"
  answer "Max =" && max(card field "RainFall") & "; Min =" &&¬
  min(card field "RainFall") & "; Ave =" && ave
  answer "Average deviation =" && sum/num
end mouseUp
```

4. Click the Rain Data button.

Result:

An answer box says Max = 20.00; Min = 10.00; Ave = 15.00. After you clear the box, a second answer box states Average deviation = 2.44.

This shows one typical use for the abs(), min(), max(), and average() functions: providing a statistical summary of data. Note that we compute the average rainfall (ave) once, then use ave in the loop. This is preferable to using average(card field "RainFall") - item n of card field "RainFall" inside the loop because the latter approach requires HyperTalk to recompute the average for each cycle of the loop, thus slowing down the script. The deviation calculation measures how much the monthly rainfall is likely to differ from the average. It averages the difference between actual rainfall and average rainfall. Therefore, this example suggests that the average April rainfall is 15 inches and that most likely the actual value for a given year would be within about 2.5 inches of that.

Initializing Variables in openStack

To initialize special variables that are going to be used in your math functions and formulas, you should put the assignment of these variables into the openStack handler. You also must put something into var to indicate that the variables have been initialized. Suppose that you want to guarantee that the variables are set only when the stack is first opened. This is important, because often a sophisticated user opens a stack, leaves it, and then returns. If the user expects the stack to be as it was left, the vari-

ables must not be reset. Here is a script to ensure that variables are not reset after the stack is opened.

```
on openStack
   global varsInited
   if varsInited is not true then
     <initialize variables here...>
     put true into varsInited
   end if
end openStack
```

Review: The *random()* Function

This is another function covered in chapter 5, "Simple Math." Its syntax is as shown in Figure 13-5.

```
the random of <integer>
random(<integer>)
```

Figure 13-5. The `random()` function syntax.

When called, the `random()` function returns a random number in the range 1 to `<number>`. What if you really want a number in the range 0 to 1? Then, you can use the expression `random(2) - 1`. The function yields numbers in the range 1-2, so subtracting 1 gives the range 0-1. Suppose you want numbers in the range 0.00 to 0.99. Because `random()` returns integers, you can't do this directly, but you can do this: `(random(100) -1)/100`. The function `random(100)` returns numbers in the range 1 to 100. Subtracting 1 makes the range 0 to 99, and dividing by 100 brings that down to 0.00 to 0.99.

The *sqrt()* Function

The `sqrt()` function returns the square root of its argument; that is, it returns a number that, when multiplied by itself, yields the original value. The syntax is shown in Figure 13-6.

```
the sqrt of <number>
sqrt(<number>)
```

Figure 13-6. The `sqrt()` function syntax

Here `<number>` should be 0 or greater. The square root of a negative number is imaginary, and that's one realm into which HyperTalk does not venture.

Example 3:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Enter in the Message box

```
sqrt(169)
```

Result:

The Message box displays `13`, the square root of 169.

3. Enter in the Message box

```
sqrt(-169)
```

Result:

The Message box displays `-NAN(001)`.

That doesn't look like a HyperTalk response and it isn't. HyperTalk uses math functions built into the Macintosh. They are part of the Standard Apple Numerics Environment, called SANE for short. And `-NAN(001)` is an error message returned by this system in response to our improper use of `sqrt()`; you may recall that NAN stands for not a number.

Square roots show up in many places. For instance, suppose that you're standing by the ocean and see a ship just on the horizon. How far away is it? More generally, how far is the horizon as a function of how far you are above the surface? We can draw a picture (see Figure 13-7) to represent this situation.

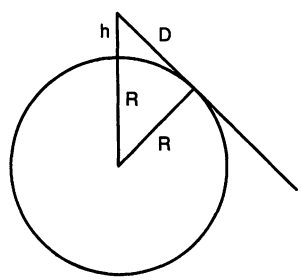


Figure 13-7. The distance to the horizon.

Here R is the radius of the Earth, h is how high your eyes are above sea level, and D is the distance from your eyes to the horizon. A little geometry and algebra yields this equation:

$$D = \text{sqrt}(2Rh + h^2)$$

Normally, R is much bigger than h, so you can neglect the h^2 term, but because the Mac's going to do the calculation, leave the term in. If you give h in feet, R should be in feet. That gives dimensions of feet squared, but taking the square root gets you back on your "feet" again. Then, you can divide by 5280 feet/mile to get the horizon distance in miles.

Example 4:

1. Go to the Math Card of your HyperTalk Bible stack.
2. Create a button called `Horizon` and give it this script:

```
on mouseUp -- Horizon button
  ask "How many feet high?"
  if it is empty then exit mouseUp
  set the numberFormat to "0.0"
  put sqrt(41805000*it + it*it)/5280 into distance
  answer "The horizon is" && distance && "miles distant."
end mouseUp
```

3. Click the button and respond to the Dialog Box question with `5`.

Result:

You are informed `The horizon is 2.7 miles distant.`

The number 41805000 is the Earth's average diameter (2R) in feet. We used the `numberFormat` property (see chapter 5, "Simple Math") to show the result to the nearest tenth of a mile.

This is a nice button. Use it and you find that your horizon distance at a height of 50 feet is 8.7 miles and at a 100 feet is 12.2 miles.

Financial Functions

As your financial life grows more complex, you may find help in the two functions `compound()` and `annuity()`. The `compound()` function is the simpler, so let's look at it first.

The *compound()* Function

Most establishments today pay compound interest on investments. If you're a little hazy about what that means, here's a quick summary. Suppose that you place $100 in an interest-earning account. This investment is called the principal. Suppose that the investment pays 10 percent compound interest yearly. The first year, you get 10 percent of the $100, or $10. The next year, you get 10 percent of the original principal and 10 percent of the interest you earned. That is, you get 10 percent of $110, or $11. The third year, you get 10 percent of $121, and so on. This contrasts with simple interest, in which you get 10 percent of the original principal every year. The difference between compound interest and simple interest after two payments is just $1 after two years for this example, but with longer investments the difference can be substantial.

The `compound()` function calculates compound interest values; it takes two arguments as shown in Figure 13-8.

The first argument is the interest rate expressed as a decimal. That is, 10 percent is expressed as 0.10. The second argument is the number of times the interest is compounded. The function returns the net value of a unit investment (i.e., $1) at the

end of that time period. To get the actual value, multiply this result by the original principal. Let's try it.

```
compound(<rate>,<times>)
```

Figure 13-8. The `compound()` function syntax

Example 5:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Enter in the Message box

```
compound(0.10,10)
```

Result:

The Message box displays `2.593742`.

This means, for example, that if you invested $100 at 10 percent interest compounded yearly for 10 years, your investment would be worth $100 * 2.593742, or $259.37. Because $100 dollars of that is your original investment, your interest earnings are 159.37. At 10 percent simple interest, you would have earned $10 interest for 10 years, or $100, so compounding the interest makes a big difference here.

Most banks compound more than once a year. To find an effective yearly rate, we divide the yearly interest rate by the number of compoundings per year to get the interest rate per compounding. Then, we compound for one year to get the net effect. Interest rates change and so do the number of times interest is compounded yearly, so let's make a button that calculates the net yearly interest rate for various combinations. Most people prefer to enter interest rates as percents rather than decimals, so we'll let the button make the conversion.

Example 6:

1. Go to the Math Card of your HyperTalk Bible stack.
2. Create a button called `Compound Interest` and give it this script:

```
on mouseUp -- Compound Interest button
  ask "Enter yearly interest rate as a percent:"
  if it is empty then exit mouseUp
'  put it/100 into rate -- convert % to decimal form
  ask "Enter number of compoundings per year:"
  if it is empty or it is 0 then exit mouseUp
  set numberFormat to "0.000"
  answer "That's a net yearly rate of" &&¬
  (compound(rate/it,it) - 1) * 100 & "%" --convert to %
end mouseUp
```

3. Click the Compound Interest button and respond to the dialog windows. Enter 10 for the interest rate and 4 for the number of compoundings.

Result:

An answer box states `That's a net yearly rate of 10.381%`.

The expression `rate/it` converts the yearly interest rate to the rate for the compounding period. Then, the function returns the final value: `1.10381` in this case. We subtract 1 (the original principal) to get amount earned in interest: 0.10381, in this example. Because this is the amount of interest earned on $1, the interest rate is $^{0.10381}/_{1}$, or just 0.10381. Finally, we multiply it by 100 to convert to the percent format.

It's interesting to find the equivalent rates for compounding periods of monthly, daily, and hourly (8760 hours per year). Check it out.

***compound* Function Formula**

The `compound()` function uses a standard formula to calculate its return value: `compound(rate,times) = (1+rate)`^{times}.

The *annuity()* Function

First, let's discuss what an annuity is. When you purchase an annuity from, say, an insurance company, they promise to pay you back a fixed amount yearly for a given number of years. The tricky part is that they pay interest into your account in an amount based on your current balance. You put in X dollars to start. Each year, they diminish your account by the dollars they pay you, and they increase your account by the interest payments they pay in to the annuity. The payment amount is chosen so that your account reaches zero after the last arranged payment. The net result is that you get back more money than you put in because of the interest payments. It's pretty much a mortgage in reverse. In a mortgage, the amount you owe is decreased by your payments and increased by the interest on the balance, and the payment size is gauged so that the balance reaches zero at the last payment.

HyperTalk's annuity function is designed to help you calculate the cost of buying an annuity given the size of payment you want, the interest rate, and the number of payments. The syntax is shown in Figure 13-9.

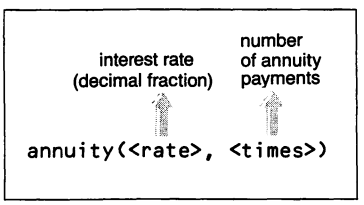


Figure 13-9. The `annuity()` function syntax.

The function takes two parameters: the interest rate, expressed as a decimal, and the number of payments. It returns how much money you must put in now to get back a unit payment, that is, $1 per payment if you are using dollars. Multiply this value times the desired payment to get how much money the annuity will cost you now.

Example 7:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Enter in the Message box

```
annuity(0.08,10)
```

Result:

The Message box displays `6.710081`.

This means that if you pay $6.710081 for an annuity of this type, you will receive yearly payments of $1 for 10 years. If you want a little more per year, say $20,000 a year for 10 years, the annuity will cost you $20,000 times 6.710081, or $134,201.62. Note that your return is 10 x $20,000, or $200,000, so you get back nearly $66,000 more than your original investment. Of course, inflation would decrease the buying power of those future payments.

Now, use `annuity()` in a field script to calculate the purchase prices for annuities with different payout periods.

Example 8:

1. Go to your HyperTalk Bible stack and create a new card called `Annuities`.
2. Create a card field called `Annuities`. Size the field so that it is near full width. Select the Lock Text feature and put the following text into the card script:

```
on mouseUp
  ask "Payment desired?"
  if it is empty then exit mouseUp
  put it into payment
  ask "Interest rate in percent?" with "10"
  if it is empty then exit mouseUp
  put it into rate
  set numberFormat to "0.00"
  put "rate =" && rate & "%; payments = $" & payment & return¬
  & return into card field "Annuities"
  divide rate by 100 -- convert to decimal rate
  repeat with n = 2 to 5
    put n*5 into paytimes
    put payment * annuity(rate,paytimes) into cost
    put payment * paytimes into payback
    put "number of payments =" && paytimes & "; cost = $" &¬
    cost & "; payback = $" & payback & return after card¬
    field Annuities
  end repeat
end mouseUp
```

3. Click the mouse in the field.

Result:

You are asked for a payment size; enter `20000`. Then, you're asked for an interest rate; enter `8`. Costs for four different payment periods are shown in the field, as illustrated in Figure 13-10.

Naturally enough, the longer you want to get payments of $20,000 a year, the

more the annuity costs, but a 20-year annuity is less than twice the cost of a 10-year annuity. That's the effect of being paid interest over a longer interval.

```
rate = 8%; payments = $20000

number of payments = 10.00; cost = $134201.63; payback = $200000.00
number of payments = 15.00; cost = $171189.57; payback = $300000.00
number of payments = 20.00; cost = $196362.95; payback = $400000.00
number of payments = 25.00; cost = $213495.52; payback = $500000.00
```

Figure 13-10. Annuity purchase costs.

You might be more interested in how much payback you get for a fixed investment. We have

```
investment = annuity() * payback
```

so it's easy to solve for the payback:

```
payback = investment / annuity()
```

Running through a few different values for the payback time for a fixed investment, you find you get more money back the longer you contract for, but, of course, you get less per year.

***annuity()* Return Formula**
HyperTalk uses the following formula to calculate the return value for annuity: `cost = (1 -(1+rate)(-times) )/rate`. Here `rate` is the interest rate as a decimal, `times` is the number of payments, and `cost` is the cost of an annuity paying out $1 per payment.

Mortgages

Mathematically, annuities and mortgages are the same thing. The practical difference is that with an annuity, you essentially loan a company a lump sum of money that they then pay back to you in installments, whereas, with a mortgage, a company loans you money that you then pay back to them in installments. The difference is just the point of view. That which is a mortgage to you essentially is an annuity to the lender. This equivalence means you can use the `annuity()` function to make mortgage calculations.

For mortgages, as with annuities, the given information is an interest rate and a

number of payments. You can use the `annuity()` function to connect the size of the payments to the size of the loan:

loan size = annuity(rate,times) * payment

If you know the loan size, the interest rate, and the number of payments, you can calculate the payment size.

Trigonometric Functions

Trigonometry is handy for figuring geometric relationships, and the functions themselves have their own beauty. The three primary trigonometric functions are the sine, cosine, and tangent functions. HyperTalk refers to them by their traditional abbreviations: `sin`, `cos`, and `tan`. Figure 13-11 shows the syntax.

```
sin(<angle>)
cos(<angle>)
tan(<angle>)
```

Figure 13-11. The trigonometric functions syntax.

These functions normally are defined in terms of the ratios of the sides of a right triangle. Suppose that we call one of the acute angles A, the side opposite it a, the short side adjacent to it b, and the long side (the hypotenuse) h, as shown in Figure 13-12.

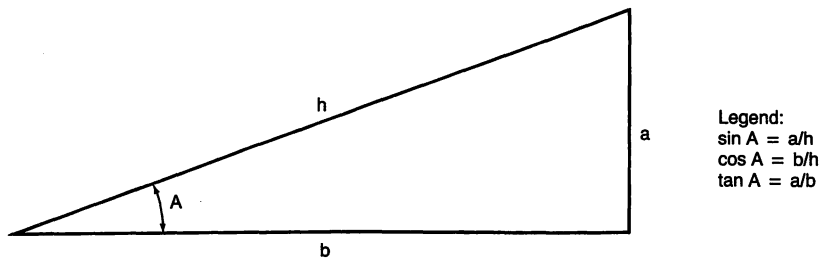


Figure 13-12. A right triangle.

The trigonometric functions are defined as follows:

```
sin(A) = a/h
cos(A) = b/h
tan(A) = a/b
```

These definitions imply that the value of a sine or cosine lies between -1 and 1, but the value of the tangent can vary from $-\infty$ to ∞.

Note, however, that the HyperTalk syntax requires that the angle be measured in a unit called the *radian*. If you use degrees, you will get wrong values. Fortunately, there is a simple conversion:

$$\pi \text{ radians} = 180^\circ$$

To convert radians to degrees, therefore, multiply by $^{180}/_{\pi}$. To convert from degrees to radians, multiply by $^{\pi}/_{180}$. That is, if D is an angle in degrees, use `D * pi / 180` for `A`, the angle in radians. Here, pi is the HyperTalk constant representing the numeric value of π.

Once you know any one side and any one angle (in addition to the right angle) in a right triangle, you can use these functions to find the value of the third side. For example, suppose that you want to build a wood ramp that rises 5 feet to the top of a wall and has a slope of 20°. (See Figure 13-13).

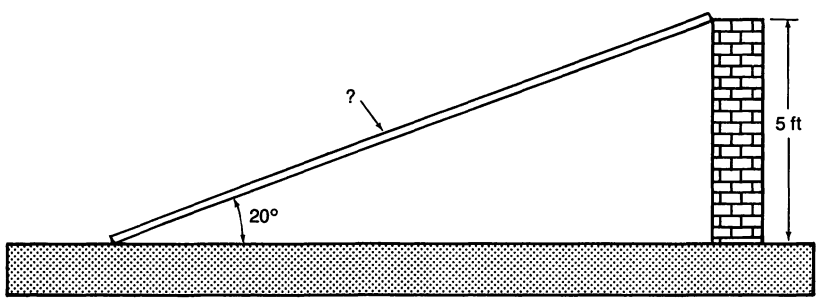


Figure 13-13. A ramp.

How long must the ramp be? In our diagram, the 5 feet is a, the 20° is A in degrees, and the length of the ramp is h. In radians, A is $20 * {}^{\pi}/_{180}$, or $^{\pi}/_{9}$. So we have $\sin({}^{\pi}/_{9}) = {}^{5}/_{h}$. Solving for h yields the value $^{5}/_{\sin(\pi/9)}$. (See Figure 13-14.)

Now we can go to the Message box and calculate h.

Example 9:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Enter in the Message box

```
5/sin (pi/9)
```

Result:

The Message box displays `14.619022`.

Now, you know you should cut your planks exactly 14.619022 feet long. (For

$$\sin(\pi/9) = \frac{5}{h}$$

$$h * \sin(\pi/9) = \frac{5}{\cancel{h}} * \cancel{h}$$

$$\frac{h * \cancel{\sin(\pi/9)}}{\cancel{\sin(\pi/9)}} = \frac{5}{\sin(\pi/9)}$$

$$h = \frac{5}{\sin(\pi/9)}$$

Figure 13-14. Some simple algebra.

those of you without decimal rulers, 14 feet, 7 $^{27}/_{64}$ inches probably would be close enough.)

Similarly, you could use $^{5}/_{\tan(\pi/9)}$ to calculate the horizontal distance from the start of the ramp to the base of the wall; that turns out to be about 13.74 feet.

You don't like converting to radians? In the next chapter, we show you a way to have the stack do it for you.

The Look and Feel of the Sine Function

Although the trigonometric functions were developed in the context of right triangles, they're much more general than that. Many things—musical tones, mechanical vibrations, electrical oscillations, circular motion, and more—can be described using these functions. We look at an example soon, but first let's look at the functions themselves.

Figure 13-15 portrays the sine function in an x-y plot. Note that the sine function repeats itself. Mathematicians call this a *periodic* function. The sine function has a period of 360°, or 2π radians, because that's how long it takes the function to repeat. We just show two cycles, but the pattern repeats forever. That's one property that makes the sine function well suited to represent oscillations, for oscillations also repeat the same pattern of motion over and over.

What does this repetition mean geometrically? Well, imagine that x represents an angle between the horizontal axis and a rotating line, as shown in Figure 13-16. As x increases, it eventually reaches 360° (or 2π radians), at which point, the line is back to its starting point. If the line continues to rotate, we say the angle continues to increase: 450°, 540°, 630°. When it reaches 720°, the line is back at its starting point again, so as x increases, the pattern repeats over and over.

A second important point about the sine function is that the value of the function alternates between 1 and −1. The negative values occur for the angles between 180° and 360°. In Figure 13-11, the y-value is negative, so the definition of sine produces negative values.

It's easy to scale the sine function to other values. For example, if you want a value to vary between 100 and −100, you can use 100 * sin(t), letting t take on a range of values. If you want an x coordinate that varies from 100 to 400, you can use

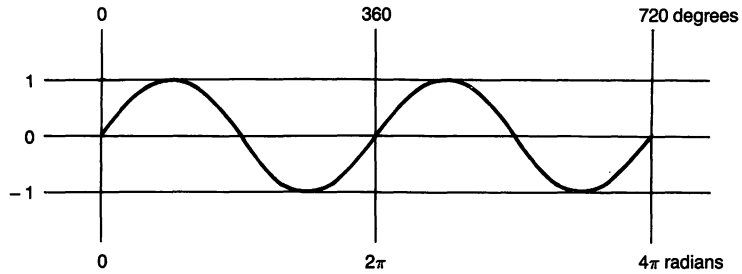


Figure 13-15. The sine function `y = sin (x)`.

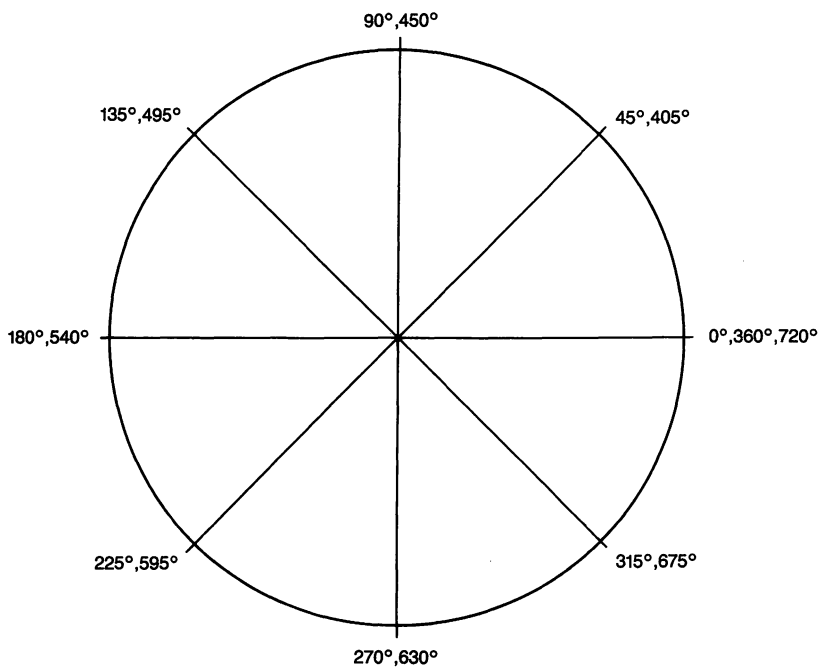


Figure 13-16. Angles bigger than 90°.

250 + 150 *sin(t). At its largest, sin(t) is 1, so the expression becomes 250 + 150, or 400. Similarly, when sin(t) becomes − 1, the expression becomes 250 − 150, or 100.

The cosine function is much like the sine function. Shift the sine function to the left by 90°, and you get the cosine graph. What about the `tan()` function? Figure 13-17 shows its graph.

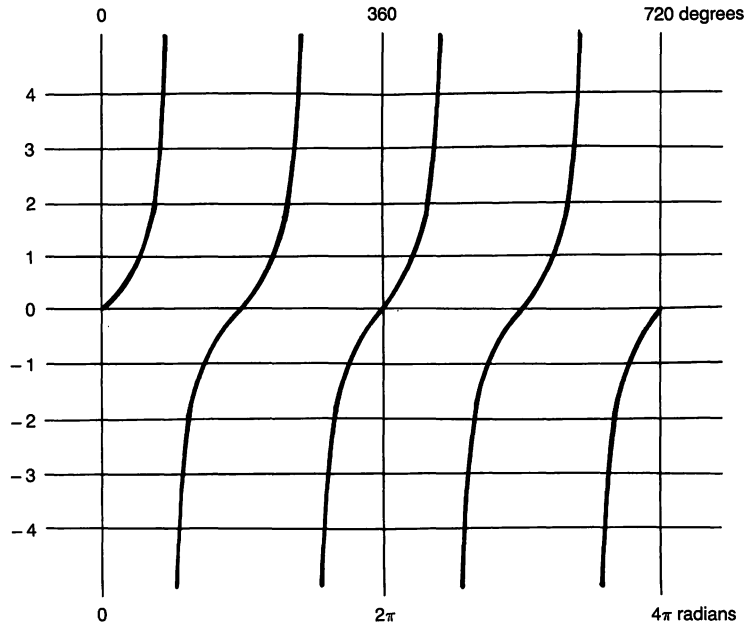


Figure 13-17. The `tan()` function.

Like the sine function, the tangent is periodic. However, it repeats every 180° rather than every 360°. Also, instead of being limited to the range −1 to 1, the tangent function varies from negative to positive infinity. (To conserve paper, we show only part of that range in Figure 13-17.) If you have a yearning for the infinite, then `tan()` may be your function.

Circular Motion

You can use the `sin()` and `cos()` functions to generate circular motion. The fundamental idea is that points on the circumference of a circle can be located by using the sine and cosine functions. See Figure 13-18.

If you have the angle and the radius of the circle, you can calculate the x and y coordinates of a point:

```
x = r cos A
y = r sin A
```

Because points on a video screen are usually specified by x and y positions, you can use the sine and cosine function to construct a circle on the screen. In fact, by letting the angle change at a steady rate, you can make an object move in a circle. That is what the next example does.

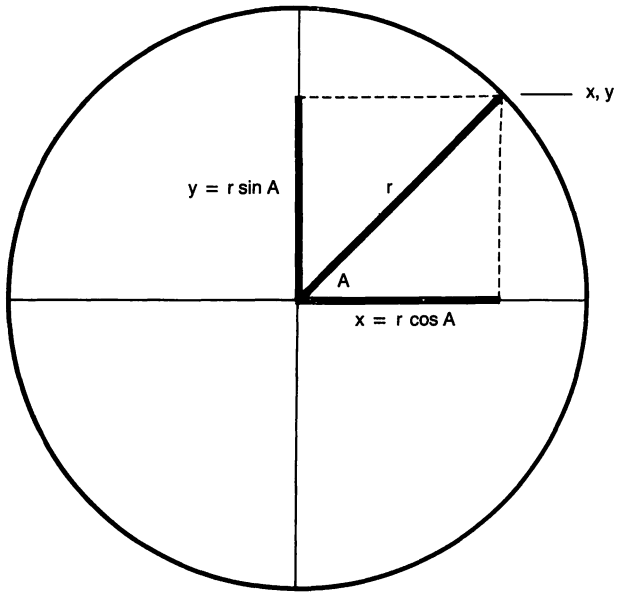


Figure 13-18. Locating points on a circle.

Example 10:

1. Go to the Math Card of your HyperTalk Bible stack.
2. Create a button called `Circle`. "Uncheck" the Show Name box. Select the Marker Cross icon (Icon ID 1018). Then, give the button this script:

```
on mouseUp -- Circle button
  put 50 into radius
  set the loc of me to 256 + radius,170 -- center at center screen
  put 1 into n
  repeat while the shiftKey is up
    put n * pi/4 into angle
    add 1 to n
    put 256 + radius * cos(angle) into x
    put 170 + radius * sin(angle) into y
    beep
    set the loc of me to round(x),round(y)
  end repeat
end mouseUp
```

3. Click the Circle button.

Result:

The button moves in a circle until you hold the Shift key down.

This script uses the sine function for one motion (the vertical movement) and the cosine function for the other (the horizontal movement). Combining these two motions produces circular motion. Using an argument of `n * pi/4` makes each step in `n`

correspond to one-eighth of a complete cycle. We put an offset of 256 into the x coordinate and 170 into the y coordinate so that the button circles around a point at the center of the card. Incidentally, because the Mac screen coordinates increase in the downward direction rather than the conventional (for math) upward direction, increasing the angle here produces clockwise rather than counterclockwise motion.

The *atan()* Function

The sine, cosine, and tangent functions let you compute one side of a right triangle when you are given another side and an angle. The `atan()` function (short for arctangent) lets you find the angle when you know the two short sides of an angle. Look back at Figure 13-19. The tangent of A was $^{y}/_{x}$. But suppose you know y and x and want A. Then, `atan(y/x)` is the angle A, in radians.

```
atan(<number>)
```

Figure 13-19. The `atan()` function syntax.

In short, you give `atan()` the numerical value of $^{y}/_{x}$, and it gives you the angle. Suppose, for example, that you notice that a 50-foot flagpole casts a 60-foot shadow. How high is the sun above the horizon? From Figure 13-20, you can see that $^{y}/_{x}$ is $^{50}/_{60}$.

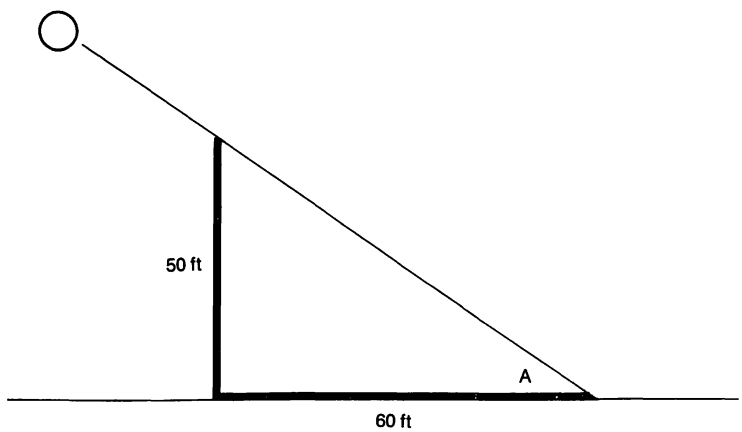


Figure 13-20. Using `atan()` to calculate a height.

The next example takes it from there.

Example 11:

1. Go to your HyperTalk Bible stack and open the Message box.

2. Enter in the Message box

```
atan(50/60)
```

Result:

The Message box shows `0.694738`, the angle in radians.

3. Enter in the Message box

```
atan(50/60) * 180/ pi
```

Result:

The Message box shows `39.805571`, the angle in degrees.

Here we made use of the fact that there are $^{180}/_{\pi}$ degrees to the radian. Note that because `atan()` returns an angle, we multiply the conversion factor times the result, not times the argument ($^{50}/_{60}$).

Exponential and Logarithmic Functions

This family of functions probably is of the most interest to scientists, engineers, and mathematicians. Both exponential and logarithmic functions regularly turn up in the mathematical representation of nature, and logarithmic functions frequently are useful in presenting data graphically. These functions also often appear in financial, statistical, and economic modeling.

The *exp()* Function

If you've had calculus, you may recall that the exponential function is the one whose derivative and integral is the same as the function itself. (If you graph a function as a curve, the derivative gives the slope of the curve, and the integral gives the area under the curve.) The equation for this function is this:

$$f(x) = e^x$$

Here e is a fundamental constant of mathematics. Like π, e is a number whose digits go on forever, never falling into a repeating pattern. Its value to nineteen decimal places is 2.718281828459045235.

HyperTalk represents this function as `exp(x)`, which, using HyperTalk's exponent operator, is equivalent to e^x. See Figure 13-21 for the syntax.

```
exp(<number>)
```

Figure 13-21. The `exp()` syntax.

In Figure 13-21, `<number>` is the exponent, and `exp(<number>)` returns the value of e raised to that power.

Example 12:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Enter in the Message box

```
exp(10)
```

Result:

The Message box displays `22026.465795`; the exponential function gets big fast!

Fractional Exponents

A fractional exponent corresponds to a root. This means, for instance, that `exp(0.5)` represents the square root of e, and `exp(3.5)` represents e cubed times the square root of e, that is e^3 * e^0.5. A negative exponent produces a value that is the inverse of the value produced by a positive exponent. That is, `exp(-5)` is the same as `1/exp(5)`.

The exponential function models many forms of growth, such as population growth. Often, however, it is more convenient to use bases other than e. For example, the compound interest formula $(1 + r)^n$ is a form of exponential growth, except that 1 + r instead of e is used for the base. Another useful base is 2.

The *exp2()* Function

The function `exp2()` is like `exp()` except that it uses 2 instead of e as a base:

$$\texttt{exp2(x)} = 2^x$$

Its syntax is shown in Figure 13-22.

```
exp2(<number>)
```

Figure 13-22. The `exp2()` function syntax.

It returns 2 to the `<number>` power.

This function is useful in the computer environment because, like the internal coding used by computers, it's based on powers of two. For instance, the number of different values that can be stored in n bits is 2^n. Thus, the number of values that can be stored in an 8-bit memory unit is `exp2(8)`, which evaluates to 256. And the number of values that can be stored in a 32-bit memory unit is `exp2(32)`, which

evaluates to 4294967296. This gives us one measure that proves a 32-bit machine, like a Mac, is more powerful than an 8-bit machine, like the original Apple.

The `exp2()` function also provides a convenient way to describe radioactive decay. A radioactive material is characterized by its halflife. After one halflife, half of a radioactive sample has decayed. After the next halflife, half the remaining sample has decayed, and so on. Thus, an element with a halflife of 50 years, has only $^1/_4$ the original amount left after 100 years. In general, if you start with M kilograms of an element having a halflife of HL years, you have M/exp2(T/HL) kilos left after T years.

The *exp1()* Function

Because `exp2(x)` returns 2 to the x power, you might expect that `exp1(x)` returns 1 to the x power. But 1 to any power is still 1, so that would be a silly function. (See Figure 13-23 for the `exp1()` syntax.) Instead, `exp1(x)` returns 1 less than e to the x. That is,

$$\text{exp1}(x) = e^x - 1.$$

```
exp1(<number>)
```

Figure 13-23. The `exp1()` function syntax.

It's simple to check that this is so.

Example 13:

Go to your HyperTalk Bible stack and enter

```
exp1(1)
```

in the Message box.

Result:

The Message box displays `1.718282`, which is $e^1 - 1$.

This function is offered as a convenience for those applications, such as blackbody radiation calculations in Physics, that require $e^x - 1$ instead of e^x.

Different Bases for Exponents

Suppose that you want to use a base besides 2 or e for exponentiation. In general, if b is a base you wish to use, such as 10 or 12, you can use `exp(x*ln(b))` to get b to the xth power. Here `ln()` is the natural logarithm function that we discuss next. In HyperTalk, it is faster to use the exponentiation operator: `b^x`.

The Logarithmic Functions

The logarithmic functions complement the exponential functions. For instance, consider the equation

$$y = e^x$$

The `exp()` function lets you calculate y if we know x. Suppose that you know y and want to find x? Then, you can use the inverse function to `exp()`. It's called the *natural* logarithm, or the logarithm base e, and is represented in HyperTalk by the `ln()` function. The nature of an inverse function is that it undoes what the original function does. That is, `ln(exp(x))` is x. Taking the natural log of both sides of the equation at the beginning of this paragraph gives this:

$$\ln(y) = x$$

If e^x is y, then `ln (y)` is x. In short, `ln(<number>)` returns the exponent to which you must raise e to get `<number>`. See Figure 13-24 for the `ln()` syntax.

```
ln(<number>)
```

Figure 13-24. The `ln()` syntax.

Example 14:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Enter in the Message box

```
ln(10)
```

Result:

The Message box displays `2.302585`. This is the natural log of 10, so e to the 2.302585 power is 10.

The `log2()` function is much like `ln()` except that it is the inverse of `exp2()`. The syntax for the `log2()` function is shown in Figure 13-25.

```
log2(<number>)
```

Figure 13-25. The `log2()` function syntax.

The return value is such that 2 to that power is `<number>`. For example, `log2(8)` is 3, because 2 to the third power is 8.

You may wonder why HyperTalk calls this function `log2()` rather than `ln2()`.

The reason is that mathematicians use the notation *ln* to refer to natural logarithms, that is, to base e. But they use *log* for logarithms to other bases.

Finally, the `ln1()` function is the inverse function to `exp1()`. It's defined this way:

$$\ln1(x) = \ln(x + 1).$$

Its syntax is shown in Figure 13-26.

```
ln1(<number>)
```

Figure 13-26. The `ln1()` function syntax.

Some references describe `ln1(x)` as the natural logarithm of x plus 1, which is true as long as you understand this means adding 1 to x, then taking the log, as opposed to taking the log, then adding 1. Again, this is a function offered as convenience; you can just as easily use ln(x + 1).

Math Masters

Base 10 Logarithms

Suppose you want logarithms to some other base, such as 10. In general, to get the log of a number n to a base b, you can use the expression `ln(n)/ln(b)`. For example, the base 10 log of 2 is `ln(2)/ln(10)`, or 0.30103.

Function Behavior

The exponential and logarithmic functions, being inverses of each other, exhibit different behavior. If you increase the argument to `exp()`, its value at first increases slowly and then faster and faster. If you increase the argument to `ln(x)`, it increases relatively rapidly at first and then more and more slowly.

Many things in nature and society undergo exponential growth, at least for a while. One of the consequences of exponential growth is that, unchecked, it soon gets out of hand. A second consequence is that in real situations, eventually exponential growth does get checked. If you put rabbits in a suitable environment, the rabbit population at first tends to grow exponentially. If it were to continue, eventually you'd have every square foot of land covered with rabbits. But before that can happen, disease, overcrowding, and a lack of food supplies arrests the population growth and even cuts it back again. A business that expects to grow 20 percent a year is predicating its plans on exponential growth. Eventually, that growth rate will outstrip any possible demand, and the managers of the business better be aware of that.

Students of growth often use semilogarithmic plots of data. That means in-

stead of plotting, say, population versus time, they plot ln(population) versus time. The reason for this choice is that if the population is growing exponentially, the graph shows up as a straight line on the semilog plot. Furthermore, the slope of the line indicates the growth rate.

Summary

HyperTalk offers a varied mix of useful math functions. Some are general purpose, some are financial, and some are oriented toward science and engineering. You don't have to be restricted by those categories. The `compound()` function can be used to represent biological growth by interpreting the interest rate as a growth rate. Trig functions and log functions can be used as drawing and graphing tools.

Keep in mind, however, that HyperTalk was not designed to be a powerful number cruncher. It can perform sophisticated calculations, but not rapidly. If you want a button or card to calculate, say, mortgage information, HyperTalk is fast enough. If you want to run through ten thousand iterations of involved calculations, you are better off developing an interface between HyperCard and a Pascal or C program.

Review Questions

1. Suppose that the variable `sales` contains a list of sales figures separated by commas. Write expressions for the following:

   a. the largest sales figure
   b. the average sales figure
   c. the average value of the largest and the smallest sales figures

2. What does 20 + the random of 8 produce?
3. Suppose you place $1000 into a savings bank at 10 percent interest. Write an expression for the value of this investment

   a. after 15 years of interest compounded annually
   b. after 15 years of interest compounded quarterly
   c. after 15 years of interest compounded monthly
   d. after 15 years of simple interest

4. Which HyperTalk function can be used to calculate mortgage payments?
5. A right triangle has sides a, b, and h (the hypotenuse). Angle A is opposite side a. Match the function to the definition.

   a. sin(x)
   b. cos(x)
   c. tan(x)
   d. a/b
   e. a/h
   f. b/h

6. A right triangle has short sides of 4 feet and 3 feet. Find the angle opposite the 3-foot side.

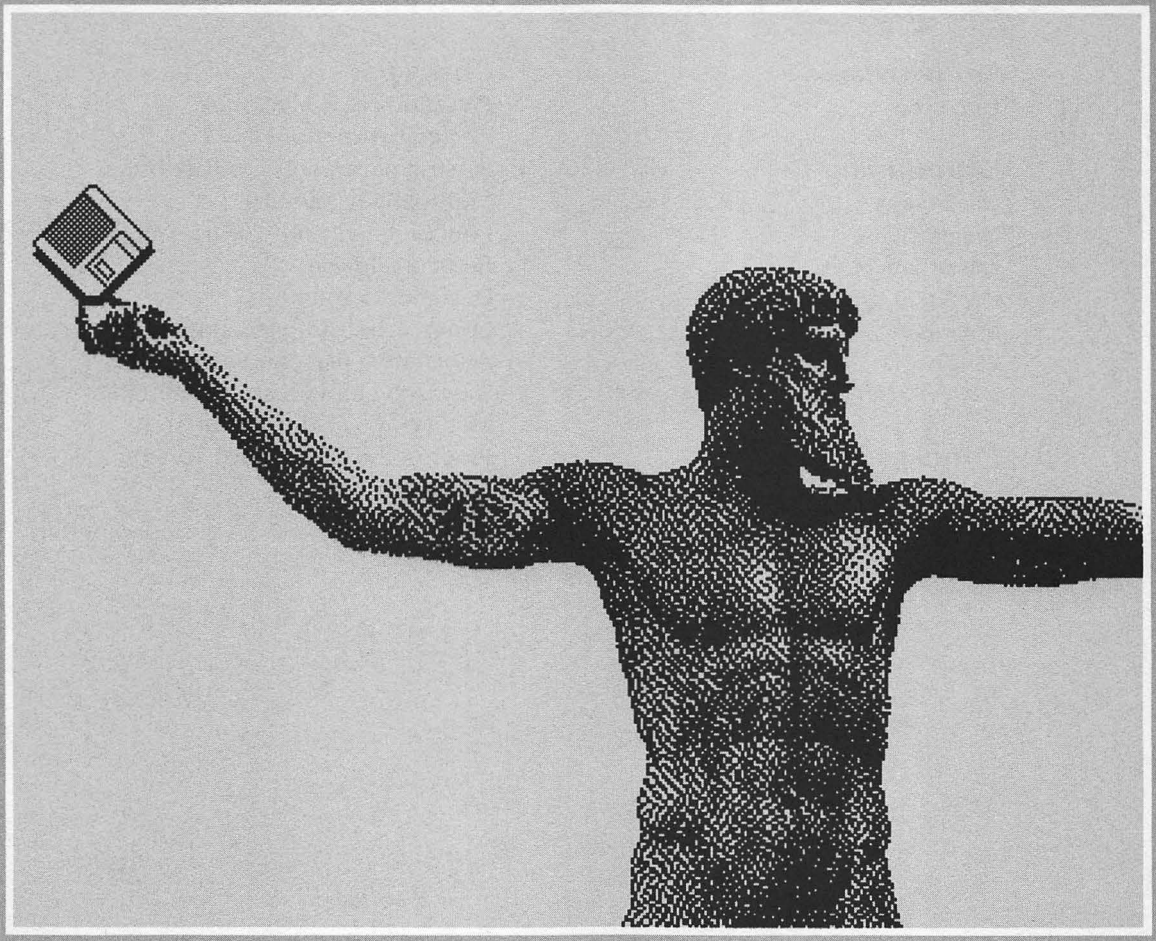
7. For each of the following functions, indicate whether it's periodic.
   a. `sin(x)`
   b. `abs(x)`
   c. `tan(x)`
   d. `exp(x)`
8. Write an expression using functions for the following:
   a. the sine of 30 degrees
   b. the 10th power of 2
   c. the cube root of e
   d. the power of 2 that 45 is

Exercises

1. Write a button script that finds the average of 50 random numbers in the range 0 to 10. What answer would you expect to get?
2. Write a button script that calculates the monthly payments and total cost of a 30-year mortgage for $100,000. Have it go from 8 percent to 11 percent in 0.25 percent increments and display the results in a card field called `Mortgage`.
3. Write a button script that moves a button in an outward-growing spiral. Hint: Modify the circling button script so that the radius increases each step instead of remaining constant.
4. Under favorable conditions, rabbits can double their population every three months. Write a button script that asks for a time in years and then shows the number of rabbits after that time. Start with two rabbits.

Chapter 14

Advanced Commands and Functions



Quick View

Keywords
`function`

Functions
`the params`
`the param of`
`the paramcount`
`the target`
`the result`
`version()`
`the version of stackDescriptor`

Concepts
User-defined functions

Concepts *cont'd*
Functions as a black box
Designer functions
Passing parameters to functions
Multiple parameters
Functions without return values
Return values
Designer commands
Commands and parameters
Alternative parameter passing
Hierarchy and custom commands
Modifying existing commands
Finding the HyperCard version
Using `the target`

Chapter 14

Advanced Commands and Functions

HyperCard and HyperTalk have energized (or re-energized) enormous numbers of computer users because they bring back the joy of tinkering. Let's face it: the Macintosh has a pretty sophisticated interface, and playing around with your own programs had become an unrewarding task unless you were an advanced progammer. Suddenly, the fun is back! You can concentrate on the creative aspects of programming rather than the technical. If you like to tinker, HyperTalk is a great tool.

In this chapter, we look at two important HyperTalk features for the do-it-yourselfer: defining your own functions and defining your own commands. Do you wish you had a function that returns the last ten characters of a string, computes the hypotenuse of a triangle, or converts scientific number notation to decimal notation? Would you like a command that shows the cards in a stack at the rate you specify? Would you like to modify the action of an existing command? We see how in this chapter, and you can use the same techniques to create functions and commands to meet your own particular needs. User-defined functions and commands are important programming features, but they also can be a lot of fun.

The actual scripting for user-defined functions and messages follows the same lines we used for button, field, and card scripts. The main difference is the interface. Typically, you invoke a button by clicking it, but to invoke a function or command, you use it in a script. Our previous scripts usually obtained information from an `ask` command or a field; functions and commands get their information from parameters. Up to now, our scripts have used the Message box or the `answer` command to report results, but user-defined functions, like system functions, use return values. We examine how to set up these new facilities, starting with functions.

User-Defined Functions Override

A user-defined function with the same name as a HyperCard function overrides the built-in function only if the user-defined message is defined at or above the point it is used. (Functions are searched for in the message hierarchy, just as are handlers.) This makes sense, because otherwise you could not modify the behavior of a HyperCard function without it potentially conflicting with a variable of the same name, nor could you override it with an XCMD or XFCN.

User-Defined Functions

In addition to supplying many built-in functions, HyperTalk lets you design your own. This is a very powerful and useful feature. With it, you can customize HyperTalk to fit your needs. Programmers often find that they need to perform some particular operation many times. For instance, you may need to find the hypotenuses of several right triangles or delete the first word from several strings. Instead of rewriting this code every time it is needed in a script, you can define a function to do the job. Then, you merely "call" the function whenever needed. This approach eases the burden on you, and it makes your scripts easier to read, because one well-named function call can take the place of several lines of more obscure code. Using functions can make your scripts look more elegant, because the details are hidden elsewhere.

Function Basics

You've already worked with several system functions, particularly in chapter 5, "Basic Math," and chapter 13, "Math Functions," but we'd like to go over the basics. In general, a function takes a list of parameters and returns a value. For example, the `length()` function takes a string as a parameter and returns a string length as a value. The `sqrt()` function takes one number as a parameter and returns another as a value. The `average()` function takes a list of numbers as a parameter and returns a number as a value.

You can think of a function as a black box that accepts input from you, does some processing, and gives you back a value. See Figure 14-1. To use a function like `length()` or `random()`, you don't need to know how it works internally. You just need to know what information to feed it and what information it returns to you.

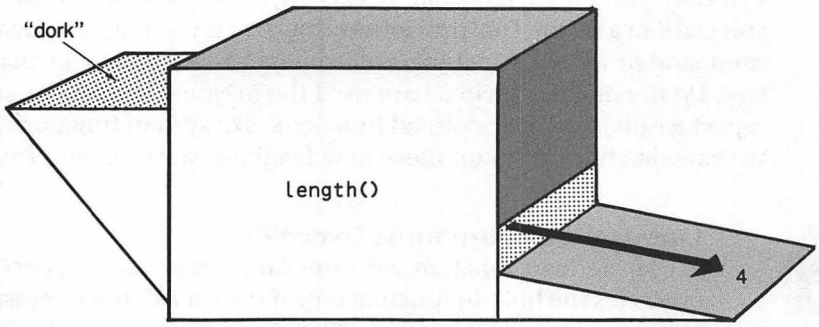


Figure 14-1. The function as a black box.

Designing a function, however, is harder (but more interesting) work than just using one. You face two different kinds of tasks in creating a function. First, you have

to work out the design specifications: What do you want the function to do? What information does it need? Then, you have to figure out the mechanics of implementation, that is, how to make it do what it's supposed to do. If this is beginning to sound a bit intimidating, don't worry, we're just talking generalities here. Function design can get involved, but often the design phase and the implementation phase take just a few seconds, so the conceptual superstructure may be imposing, but the actual work can be easy.

Designing a Function

To take advantage of the function-defining feature, you need to know how to define a function, where to place that definition, and how to call on the function.

The basic syntax for defining a function looks like Figure 14-2.

```
function <function-name>
   <command(s)>
   return <value>
end <function-name>
```

Figure 14-2. Syntax for defining a function.

Here, `function` is the HyperTalk keyword that indicates you are defining a function. The `<function-name>` parameter is the name you choose to give the function. The `<parameter-list>` part of the syntax symbolically represents the values you pass to the function. The `<command(s)>` are the HyperTalk commands you use to do the work of the function, and the keyword `return` identifies the value the function returns to you. For example, here's a function that returns the square of a number:

```
function square x
   put x * x into y
   return y
end square
```

The name of this function is `square`. The parameter `x` represents the value passed to the function. The function squares this value and puts the result into y. Then, the function returns that value. That means you can consider the function call to be replaced by the returned value in whatever expression the function call appears. For example, the command `put length("train") into railroad` is interpreted as `put 5 into railroad`, because `5` is the return value for `length("train")`.

After you have a function definition, where do you keep it? You put the definition into an object: a button, field, card, background, or stack. Once in an object, a function definition works like a message handler. That is, you can use the function in that object or in any object lower in the hierarchy. Typically, function definitions are

placed at the stack level so that all the objects in the stack can use them. If a function is particularly general, you can place it in the Home stack so that all stacks can use it.

Ways to Do a Function

You must use one of the following forms with a function: `the`, `of`, or parentheses `()`. For example, you can say `the numToChar of factor`, or you can say `numToChar(factor)`. You cannot, however, mix `the` and `()`, so `put the NumToChar(48)` is illegal, You are allowed to say

```
put the numToChar of 48
put NumToChar 48
put the numToChar of (48)
```

How do you use a function? You make a *function call.* In Figure 14-3, the `<parameter-list>` part of the syntax symbolically represents the values you pass to the function.

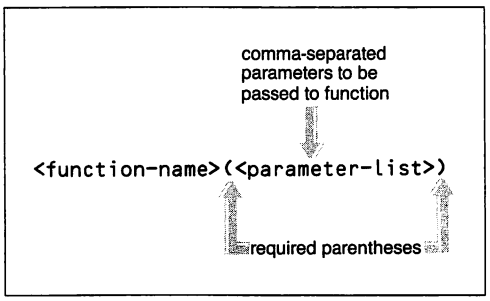


Figure 14-3. Syntax for using a function.

For example, if you want the square of 6, use the function call `square(6)`. This "calls up" the `square()` function and passes the value 6 to it. That is, the variable `x` in the definition is assigned the value 6. Note that you have to use parentheses. See Figure 14-4.

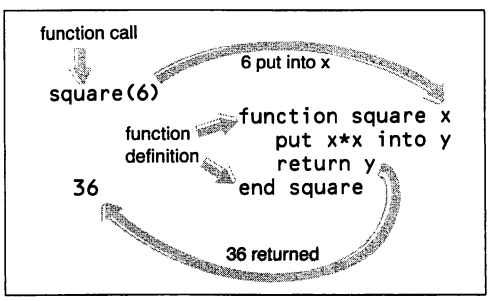


Figure 14-4. Passing a parameter to a function.

HyperTalk doesn't let you drop the parentheses, so you can't use `square 6`, even though that looks more like the definition. Also, you can't imitate the built-in functions and use the `square of 6`; user-defined functions are slightly less sophisticated than the built-in functions. Once you make the function call, the `square()` function calculates the return value of `36`, and HyperTalk uses this number in the expression containing the function call.

Often, the values passed to a function are termed *arguments;* this is the term mathematicians commonly use when discussing mathematical functions. Also, the parameter name used in the function definition may be called a *formal* parameter. A formal parameter really is just a variable used in a particular way, and passing arguments to a function really is a way of assigning values to those variables. Thus, in our example, `x` is the formal argument, `6` is the actual argument, and `6` is assigned to `x`.

Let's run through an example.

Example 1:

1. Go to your HyperTalk Bible stack and open the stack script.
2. Enter the following function definition (note that we've modified the definition slightly since you last saw it):

```
-- square(n) returns the square of n
function square num
  return num * num
end square
```

3. Return to the card, open the Message box, and enter `square(4)`.

Result:

The Message box displays the value `16`.

Here, the function call `square(4)` is replaced by its return value, `16`. Similarly, if you enter `"The square of 4 is"&& square(4)`, the Message box displays `The square of 4 is 16`. Again, the function call is replaced by the return value.

4. Enter

```
square(2 + 5)
```

Result:

The Message box displays the value `49`. 7

What happens here is that HyperTalk first figures out that `2 + 5` is `5`. Then, it passes the value `7` (and not the string `2 + 5`) to the function. Whenever you use an expression, rather than a single value, as a parameter to a function, HyperTalk first evaluates the expression and then passes the final value. The next entry carries this a step further and illustrates the flexibility of functions.

6. Enter

```
square(3 + square(2))
```

Result:

The Message box displays `49`.

First, HyperTalk passes the value `2` to `square()`. That's for the function call

inside the parentheses. Then, it adds the return value of 4 to 3 to get 7. Next, it passes the value 7 to `square()`, finally winding up with the value 49.

We modified our first definition of `square` in two ways: we replaced `x` with `num`, and we used `return num*num` instead of first calculating a y and returning it. These changes illustrate two important points about functions. First, you are free to choose the name of a parameter. Call it `x`, `num`, `MacNumber`. Whatever your choice, the formal parameter acts a variable to hold whatever value is passed. Second, the only restriction on what follows the keyword `return` is that it evaluates to a value. Instead of doing a calculation, storing it, and returning the stored value, we simply return the result of the calculation. What happens is that first the expression (here `num * num`) is evaluated, and then the final value is returned.

Because functions are replaced by their values, you can use them just like any other values in expressions:

```
repeat for n = 1 to 20
  put n && square(n) after field 1
end repeat
add ( 2 * n + square (n - 1) )/ (1 + square ( square(n) ) )
If the number of buttons < square (the number of cards) then beep
```

System Functions with Multiple Arguments
Any system function that requires multiple arguments (parameters) always requires parentheses. For example, `offset (string1,string2)` does not have a form `the offset of (string1,string2)` or even `the offset(string1,string2)`.

Multiple Parameters

HyperTalk functions can take more than one parameter. Just separate the parameters by commas. The next example illustrates the techniques for defining and using functions with more than one parameter. Given the two shorter sides of a right triangle, the function uses the Pythagorean Theorem to calculate the length of the third side, the hypotenuse. (The Pythagorean Theorem says the square of the hypotenuse is equal to the sum of the squares of the other two sides.)

Example 2:

1. Go to your HyperTalk Bible stack and open its script.
2. Enter the following function in the Message box:

```
-- hypot(a,b) returns the hypotenuse of a triangle
-- with sides a and b
function hypot x, y
  return sqrt( x * x + y * y)
end hypot
```

3. Return to the card, open the Message box, and enter the function

~~call~~ `hypot(5,12)`

Result:

The Message box displays 13, the hypotenuse of a right triangle with sides of length 5 and 12.

Note that you use commas to separate the parameters both in the definition and in the function call. As with single-parameter functions, you enclose the parameter list in parentheses in the function call, but not in the function definition. See Figure 14-5.

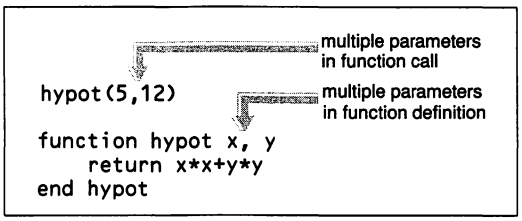


Figure 14-5. Multiple parameters.

Our `hypot()` function uses the built-in `sqrt()` function to compute the square root. Can we also employ one user-defined function inside another? Let's try it.

Example 3:

1. Go to your HyperTalk Bible stack and reopen the stack script.
2. Alter the `hypot()` definition to read as follows:

```
function hypot x, y
  return sqrt( square(x) + square(y))
end hypot
```

3. Return to the card and enter

```
hypot(5,12)
```

in the Message box.

Result:

The Message box displays the value 13.

It works. Your user-defined function can build on other functions you've defined. However, it does take the Macintosh a little extra time to call a function. As a result, this version of `hypot()` runs more than 50 percent slower than the first version. (We used the function `the ticks`, as described in chapter 12, "Date and Time," to time loops containing each version of `hypot()`.) Whether that difference is important depends on how the function is used. The difference is not really noticeable when you call the function from the Message box. If you use `hypot()` in a `repeat` command that loops a hundred times, the difference becomes apparent.

Functions with No Parameters

Some functions don't require input. Recall, for example, that the function `the date` (or the equivalent `date()`) returns the date without requiring a parameter. User functions can work the same way.

Example 4:

1. Go to your HyperTalk Bible stack, open the stack script, and add the following definition:

```
-- returns date in form day month year
function eurodate
  get the date
  convert it to long date
  return word 2 of item 2 of it && word 1 of item 2 of it &¬
  item 3 of it
end eurodate
```

2. Enter

```
the eurodate
```

in the Message box.

Result:

HyperCard complains that it `Can't understand the`.

3. Enter

```
eurodate( )
```

in the Message box

Result:

The Message box displays the date in the European format: `10 October 1988`.

This example illustrates three points. First, to define a parameterless function, just omit the parameter list from the formal definition. Second, to call the function, use the function name followed by empty parentheses. Third, as we mentioned before, you can't use the format `the functionName` with user-defined functions.

More About Ways to Do a Function

For user-defined functions, you must always use parentheses, even if there are no arguments. For example, you must write `DesignerFunction()`, not `the DesignerFunction`.

Functions, Strings, and Numbers

Ultimately, all HyperTalk function parameters and return values are passed as strings. If the string is recognizable as a number (for instance, 10 or 2.78) it can be used for arithmetic, as with our `square()` function. HyperTalk converts the string to

a numeric value when it makes calculations. You've already seen that `square(7)` works. Try `square("12")`. It also works! Indeed, the notation 12 really is shorthand for "12", the string formed from the characters 1 and 2. If you don't believe that, try typing `length(12)` into the Message box. You get the response 2 because 12 is a string two characters long.

Because function parameters and return values are strings, writing string functions is easy. Let's look at a few:

Example 5:

1. Go to your HyperTalk Bible stack, open the stack script, and add the following definition:

```
-- reverseString(str) returns a string the reverse of str
function reverseString string
  put empty into rstring
  repeat with letter = length(string) down to 1
    put char letter of string after rstring
  end repeat
  return rstring
end reverseString
```

2. Enter

```
reverseString("god pot")
```

in the Message box.

Result:

The Message box displays `top dog`.

As the name suggests, this function returns the original string in reverse order. The function call `reverseString("god pot")` assigns the string `god pot` to the variable `string`. The `repeat with` loop uses `length()` to find the number of characters in the string. Next, it puts the last character into the new string variable `rstring`. Then, the loop appends the rest of the characters to `rstring`. By using `down to`, the loop works from the end of the string to its beginning. We initialized `rstring` to `empty`, so it starts out empty before we append characters to it. Also, this ensures that `rstring` is defined, even if you skip over the loop by providing an empty string as the parameter. (If string is empty, then `length(string)` is 0. This makes the loop limits 1 to 0, so the loop is skipped.)

Of course, you can pass both numeric strings and regular strings to a function.

Example 6:

1. Go to your HyperTalk Bible stack, open the stack script, and add the following definition:

```
-- endString(nchars,string) omits first nchar chars from string
function endString nchars, string
  if length(string) < nchars + 1 then return empty
  put char nchars + 1 to length(string) of string into newstring
  return newstring
end endString
```

2. Enter

```
endString(6,"tempermental")
```

in the Message box.

Result:

The Message box displays `mental`; the first 6 characters (temper) are omitted.

The character range notation (`char nchars + 1 to length(string) of string`) provides a short cut for copying characters from one string to another. We could have used a `repeat with` loop, as we did in `reverseString()`, but that code would take more room and run about three times slower. Can we use this short cut with `reverseString`? No, because the character range notation works properly only for an increasing character count; HyperTalk doesn't support the notation `char 10 down to char 3 of string`.

You can use the same general approach to extract the beginning or middle of a string.

A Closer Look at Return Values

A HyperTalk function, like those of most other computer languages, returns a single value. With most languages, that value typically is a single number. With HyperTalk, the value is a single string. This makes the HyperTalk function more versatile. That string can represent a number, as with our `hypot()` example, or it can represent text, as with our `reverseString()` example.

There still is more you can do with the HyperTalk return value. Because you can use operators like `item` and `word` to access parts of a string, you can use a single string to hold several data items.

Consider a geometric example. If you know the hypotenuse of a triangle and the angle between the hypotenuse and one side, you can calculate the length of the other two sides by using these formulas:

$$x = h \cos(A)$$
$$y = h \sin(A)$$

Here, `h` is the hypotenuse, and `A` is the angle between side `x` and the hypotenuse. See Figure 14-6. Because HyperTalk has the `sin` and `cos` functions (see chapter 13, "Math Functions"), we can easily compute sides `x` and `y` given `h` and `A`. Now, you have a situation in which you compute two new numbers from two input numbers. To return two numbers, you can join them in a string in which the two numbers are separated by commas. Then, you can use `item` to access the numbers individually.

As you may recall from the previous chapter, there is one hitch. HyperTalk measures angles in radians rather than degrees. If you want to give A in degrees, you have to make the function convert that to radians. To do that, multiply A by π and divide by 180. The next example does just that.

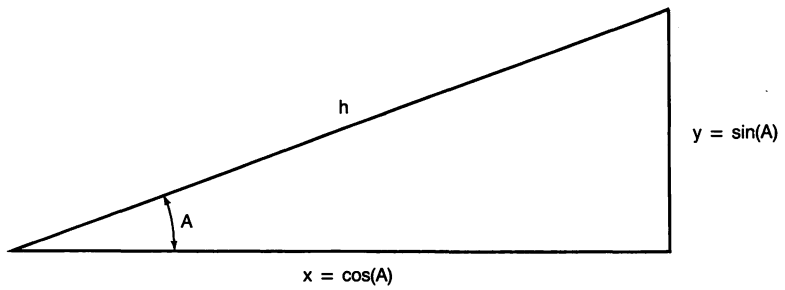


Figure 14-6. Sides of a right triangle.

Example 7:

1. Go to your HyperTalk Bible stack, open the stack script, and add the following definition:

```
-- rtSides calculates 2 sides of right triangle (x and y)
-- from the hypotenuse (h) and angle (a)
function rtSides h, a
  put h * cos ( a * PI / 180) into x
  put h * sin ( a * PI / 180) into y
  return x & "," & y
end rtSides
```

2. Enter

```
put rtSides(20, 30) into sides
```

in the Message box.

3. Enter `sides` in the Message box.

Result:

The Message box displays `17.320508,10`.

The two sides of a 30° triangle with a hypotenuse of 20 units are 17.32 (approximately) and 10 units long. The one string contains both numeric values.

4. Enter

```
item 2 of sides
```

in the Message box.

Result:

The Message box displays `10`, the length of the second side.

By using `item`, you access the individual values in the string.

5. Enter

```
hypot(sides)
```

in the Message box.

Result:

HyperTalk informs us it `Can't understand hypot`.

What's the problem? The `hypot()` function takes two values, and `sides` contains two values, so why can't you use `sides` as a parameter? The problem is that `sides` is a single string. That is, think of it as `"17.320508,10"`. As one string, it counts as one parameter. Therefore, HyperTalk assigns that whole string to the `x` variable in `hypot()` and has nothing left to assign to `y`, so the function fails. The fix is to convert the single string to two separate strings:

6. Enter

```
hypot(item 1 of sides, item 2 of sides)
```

in the Message box.

Result:

The Message box displays 20, the hypotenuse value you originally started with.

This example illustrates two important points. First, the `return` value, no matter how many parts it contains, is a single string. Second, each item in a function's parameter list is a separate string. For example, `hypot("4,3")` fails because it sends just one parameter, `("4,3")`. to the function, but `hypot("4","3")` works because it sends two parameters `("4" and "3")`.

Understanding the Use of *the*

There is a definite rule to the way that HyperTalk uses the article `the` and functions. If a function does not use parentheses or the word `of`, it must be preceded by `the`. For example, `date` does not use parentheses, so you can't say `the date`. You can only say `put the date`. Think about it. The word `the` prevents HyperCard from interpreting `date` as a variable in the statement `put date`. This concept (requiring `the` to activate functions) is intentional. It allows easy future expansion of system functions, because you don't have to worry about variable names and function names interfering with each other.

Function Parameters Revisited

The art of passing parameters to HyperTalk functions has some fine points you should know. First, as you've seen, if you use an expression, HyperTalk boils it down to a single value before passing the value along. Thus `square(5 + 2)` passes the value 7 to `square()`, and `reverseString("stop && "mug")` passes `"stop mug"` to `reverseString()`. Second, parameters are passed as strings, but if a string obviously represents a number, it can be treated numerically. Thus, `square("6")` and `square (6)` both pass the string `"6"` to the `square()` function, and the function is able do arithmetic with that value. Third, the degree of expression processing depends on whether the expressions are numeric or not. Let's take another trip to the Message box and use the system function `sqrt()` to clarify these points.

Example 8:

1. Go to any stack and open the Message box.
2. Enter in the Message box

```
sqrt("four")
```

Result:

An answer box tells us that HyperTalk `Can't understand sqrt`.

Here, the string `"four"` is passed to the `sqrt()` function. However, it is not obviously a number (at least, not to HyperTalk), so the function call fails to work. Note, however, that `sqrt(four)` does work, because the HyperTalk constant `four` represents the number `4`.

3. Enter in the Message box

```
sqrt("1" + "3")
```

Result:

The Message box displays `2`, the square root of 4.

HyperTalk recognizes the strings `"1"` and `"3"` as representing numbers, so it adds them to get 4. This is converted to string form `("4")` and sent to the `sqrt()` function for successful processing.

4. Enter in the Message box

```
sqrt("1 + 3")
```

Result:

Once again, HyperTalk says it `Can't understand sqrt`.

HyperTalk does *not* recognize `"1 + 3"` as being numeric; it sees it as a string of 5 characters: 1, space, the + character, a space, and the 3 character, so poor `sqrt()` chokes again, but not forever!

5. Enter in the Message box

```
sqrt(the value of "1 + 3")
```

Result:

The Message box triumphantly displays `2`.

Recall that the `value` function (see chapter 4, "Containers and Values") can evaluate an expression inside quotes. Clearly, you wouldn't (at least most of you wouldn't) type `sqrt(the value of "1 + 3")` when you could type `sqrt(4)` (or, for that matter, type 2), but you might want to use `sqrt(the value of problem)`, where `problem`, say, is a string formed by joining arithmetic expressions under control of a program.

User-Defined Commands

You can define your own commands much as you define your own functions. The basic syntax looks like that shown in Figure 14-7.

If you think that looks just like a handler for a system message, you're right! You define a command by creating a handler for a message by that name. The handler defines the effect of the command. As with user-defined functions, the scope of a user-defined command depends on where in the HyperCard hierarchy you place

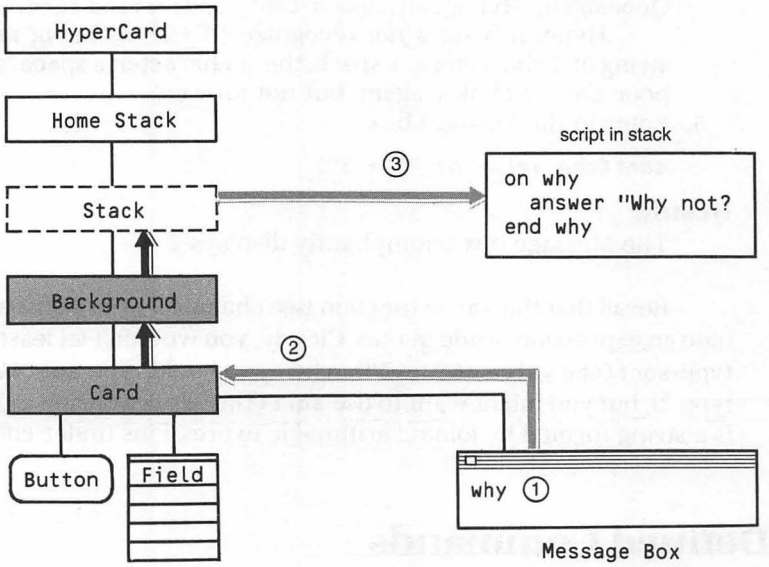
```
on <command-name>
   <command(s)>
end <command-name>
```

Figure 14-7. Command-defining syntax.

the handler. If you place it into a button, you can use the command in that button. If you place it into the stack, then all the objects in the stack can use it.

To use a command, just use its command name as a command statement. That generates a message which travels through the hierarchy until it finds a handler for that message. If you define the command on the stack level, for instance, you can access it by typing the name into the Message box or by using it in a script in a button, field, card, background, or stack. See Figure 14-8.

Suppose that you call your command `lawsuit` and that a button script includes a `lawsuit` line. When HyperTalk sees that `lawsuit`, it "knows" from context that `lawsuit` is not a variable, so HyperTalk sends a `lawsuit` message. The `lawsuit` message moves down the hierarchy just like any other message. When message reaches



Legend:
① User types command in the Message box.
② The `why` command message enters the hierarchy.
③ The `why` handler in the stack intercepts the message and handles it.

Figure 14-8. User-defined command scope.

the stack level, it finds the handler, and then HyperTalk follows the instructions there. If you don't have a handler, the message keeps going until it reaches the HyperCard level. At that point, HyperTalk realizes it hasn't found a handler yet and then tells you it didn't understand `lawsuit`.

Making and Using a Simple Command

Look at an example. We create a handler for a `slowShow` message. This handler shows the cards in a stack at a more leisurely pace than `show all` cards does.

Example 9:

1. Go to your HyperTalk Bible stack and create a card called `Custom Card`.
2. Open the HyperTalk Bible stack script and enter the following handler:

```
on showSlow
  if the number of cards is 1 then      -- a one-card stack
    play "boing"                     -- just protest mildly
    answer "Need at least 2 cards"   -- explain problem
    exit showSlow                       -- and quit
  end if
  put the name of this card into startcard -- starting point
  visual effect checkerboard to black
  go to next card
  repeat while the name of this card <> startcard
    wait 1 seconds
    visual effect checkerboard to black
    go to next card
  end repeat
end showSlow
```

3. Open the Message box and enter `showSlow`.

Result:

HyperTalk shows you all the cards in the stack, taking one second per card. Because `showSlow` is defined in the stack, you can use it from any of the stack objects.

4. Create a new button called `Show Slow`, and give it the following script:

```
on mouseUp -- Show Slow button
   showSlow
end mouseUp
```

5. Click the Show Slow button.

Result:

HyperTalk shows all the cards in the stack, taking one second per card.

Clicking the mouse sends the `mouseUp` signal to the button. There, the mouseUp handler sends the `showSlow` message. Following the usual message path, this message goes first to the card, then to the background, and then to the stack, where it activates the `showSlow` handler. See Figure 14-9.

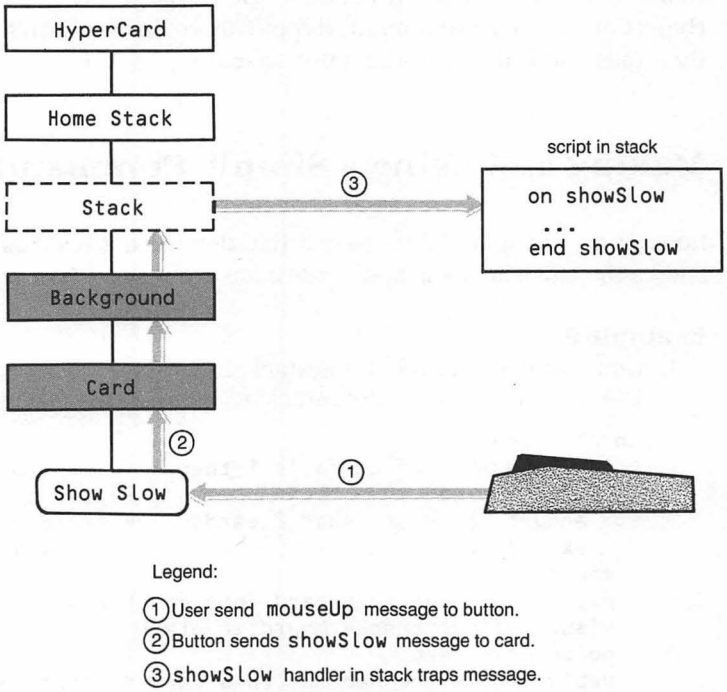


Figure 14-9. Messages on the move.

Commands with Parameters

You can set up your command handlers so that they can accept parameters. The simplest technique is the same one we used for functions: In the definition, follow the command name with a parameter list, as shown in Figure 14-10.

Let's clarify this approach with an example. We modify `showSlow` so that it takes a parameter specifying the number of seconds between card changes.

```
on <command-name> <parameter-list>
  <command(s)>
end <command-name>
```

Figure 14-10. Command handler with parameter list.

Example 10:

1. Go to your HyperTalk Bible stack and open the stack script.
2. Modify the `showSlow` handler so that it reads as follows:

```
on showSlow delay -- delay is the parameter
  if the number of cards is 1 then
    play "boing"
    answer "Need at least 2 cards"
    exit showSlow
  end if
  if delay is empty then put 1 into delay -- default value for delay
  put the name of this card into startcard
  visual effect checkerboard to black
  go to next card
  repeat while the name of this card <> startcard
    wait delay seconds -- use delay for waiting period
    visual effect checkerboard to black
    go to next card
  end repeat
end showSlow
```

3. Go to the Message box and enter

```
showSlow 3
```

Result:

HyperTalk cycles through the deck, spending three seconds on each card.

Note that we pass the command parameter by typing it after the command name. This is different from function usage, in which we must enclose the parameter in parentheses. As with functions, any expression you use as a parameter is evaluated to a single value before being passed along with the message, so you can give commands like `showSlow 2 + 3` to get a five-second delay.

Besides adding a parameter list to the handler and replacing `1 seconds` with `delay seconds`, we added the following line:

```
if delay is empty then put 1 into delay -- default value for delay
```

This chooses a value of 1 for `delay` in case the user omits a parameter when using the command. Having a script set defaults is usually a good idea. For instance, the Show Slow button, which just has the command `showSlow`, still works with the new handler. In this case, `delay` is empty, so the handler resets it to 1 second.

Multiple Parameters

To pass more than one parameter to a customized command, add a parameter list to the handler, using commas to separate the parameters. Similarly, when invoking the command, use a comma-separated list of values.

Example 11:

1. Go to Custom Card in your HyperTalk Bible stack and open its stack script.
2. Add this handler:

```
on remove wrd, fld
  repeat with n = the number of words in card field fld down to 1
    if wrd is word n of card field fld then delete word n of¬
    card field fld
  end repeat
end remove
```

3. Create a card field called `Stuff` and type the following into the text field:

```
The first the and the second the precede the last the
```

4. Open the Message box and enter the command

```
remove "the","stuff"
```

Result:

Beginning with the final `the`, each `the` in the card field Stuff is deleted.

5. Enter in the Message box

```
remove ("the","stuff")
```

Result:

Your Macintosh reports `Expected ")" but found,`.

In Step 4, `the` was assigned to `wrd`, and `stuff` was assigned to `fld`. Note in Step 5 that you can't use function-style notation for multiple parameters to commands.

To make the command more reliable, you can have the handler display an answer box if parameters are missing; just insert these lines at the beginning of the handler:

```
if wrd is empty or fld is empty then
   answer "Use this way: remove <word>,<fieldname>"
   exit remove
end if
```

Anticipating possible ways a command might be misused makes for "healthier" commands.

Alternative Method for Processing Command Parameters: *the params* and *the param of*

HyperTalk has a second method for accessing parameters passed along with a command message. It's more awkward than the one we just used, but it's also more powerful. The method uses the three HyperTalk functions `the params`, `the param of`, and `the paramcount`. When you use them, you omit the parameter list from the command handler. Instead, you call on these functions from inside the handler.

The function `the params` returns the entire command line, name and parame-

ters, that was sent to the handler. It has a simple syntax, which is shown in Figure 14-11.

```
the params
params()
```

Figure 14-11. The `the params` function syntax.

Like most HyperTalk functions that take zero or one parameter, you have your choice of a `the` form or a `()` form.

To see it work, make a simple command handler.

Example 12:

1. Go to your HyperTalk Bible stack and open the stack script.
2. Insert this handler:

```
on paramtest
  answer the params
end paramtest
```

3. Enter

```
paramtest 2,4,cow
```

in the Message box.

Result:

An answer box shows `paramtest "2","4","cow"`.

The function `the params` returns the entire line you typed. Also, each parameter has been placed in quotes. Once again, you see that HyperTalk passes parameters as strings. Also, like any other function that returns a string, `the params` returns a single string. In this case, the string contains quote marks inside the string to mark the individual parameters.

4. Enter in the Message box

```
paramtest pi
```

Result:

The answer box displays `paramtest "3.14169265358979323846"`.

5. Enter in the Message box

```
paramtest "fat" & "her", the date
```

Result:

The answer box displays `paramtest "father","8/12/88"`. (The date you see most likely will be different than the one we show.)

The last two steps show that, as with the other method for passing parameters, expressions are evaluated to single values before being passed along.

Usually, you are more concerned with knowing what a particular parameter is rather than with the whole parameter string. The `param of` function returns the

value of a individual parameter. Just tell it which parameter you want. Figure 14-12 shows its syntax.

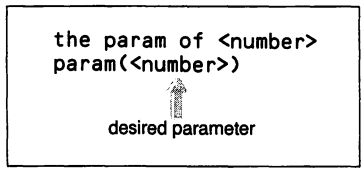


Figure 14-12. The `the param of` function syntax.

Again, HyperTalk offers two choices for the format. Both `the param of 1` and `param(1)` return the first parameter. Let's alter `paramtest` to see how `the param` works.

Example 13:

1. Go to your HyperTalk Bible stack, and open the stack script.
2. Modify the handler for `paramtest` to read as follows:

```
on paramtest
  answer the param of 1 && param(0)
end paramtest
```

3. Enter

```
paramtest "Goofy"
```

in the Message box

Result:

An answer box displays `Goofy paramtest`.

No

paramtest Goofy

First, note that you don't have to use a parameter list in the handler; The beginning line is just `on paramtest` without a list of parameter names following. Second, the zeroeth parameter is the command name itself. See Figure 14-13. Third, Goofy is not in quotes because it now is a whole string rather than a string inside a string, the way it was for `the params`. (When there is a string inside a string, HyperTalk uses quotes to show where the individual strings are.)

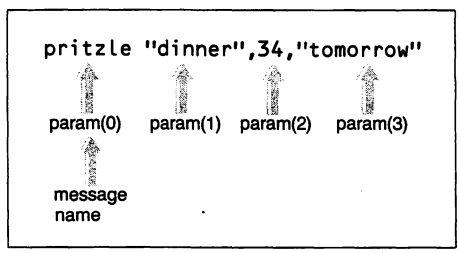


Figure 14-13. Parameters in a command line.

To pass a phrase as a single parameter, use double quotes. For instance, the command `paramtest "hot stuff"` produces the output `hot stuff paramtest`.

The final pair of this function triology is `the paramcount`, and it returns the total number of parameters (not counting the command name) passed to the current handler. It lets you design a command that takes a variable number of parameters, because the command can use `the paramcount` to tell it how many parameters to process. Figure 14-14 shows its syntax:

```
the paramcount
paramcount()
```

Figure 14-14. The `the paramcount` function syntax.

Let's write a command that can take a variable number of parameters.

Example 14:

1. Go to your HyperTalk Bible stack and open the stack script.
2. Add the following script:

```
-- vfx command: executes visual effects specified as parameters
on vfx
  repeat with n = 1 to the paramcount
    visual effect param(n)
  end repeat
  go to this card
end vfx
```

3. Enter

```
vfx "checkerboard to black"
```

in the Message box.

Result:

HyperTalk executes a checkerboard-to-black visual effect.

4. Enter

```
vfx "checkerboard to black", " dissolve to gray"
```

in the Message box.

Result:

HyperTalk executes a checkerboard-to-black visual effect followed by a dissolve-to-gray visual effect.

Step 3 used one parameter, so one visual effect was executed; Step 4 used two parameters, so two visual effects were executed.

The neat thing about this example is that it can handle any reasonable number of parameters. By using `the paramcount`, you don't hard code a particular number of

parameters into the handler the way you do when using a fixed parameter list. (That is, if you use a `repeat with n = 1 to 2` loop, the command would handle two visual effects, but not one or three.) It's this aspect of `the param` family of functions that makes them more powerful than the formal parameter list approach.

The Hierarchy

As you write your own commands, you have to decide which is the best level in the HyperCard hierarchy for them. And you may encounter the need to communicate between different levels of the hierarchy. The best way to see what this means is to work with an example. Let's start with a modest objective: making your Macintosh play the boing sound each time you open a card in a particular stack. The script is not overly complex:

```
on openCard
  play "boing"
end openCard
```

The question is where to place this script. One solution is to place it into each and every card in the stack. That certainly works, but it can be a lot of effort. It also clutters the stack.

The next level you could use is the background. If all the cards have the same background, this works great (with one proviso we get to soon). If you have two or more backgrounds, however, you have to put the script into each background to reach every card. If you just wished to sound the boing for some cards, however, the background is the best choice. You can use that background for the cards to be affected, and use other backgrounds for the remaining cards.

Assuming that you want to accompany all cards with a boing, it's simplest to go to the stack level and to intercept the `openCard` message there. Try it.

Example 15:

1. Go to your HyperTalk Bible stack and open the stack script.
2. Add this handler:

```
on openCard
  play "boing"
end openCard
```

3. Return to the card level and move through the cards.

Result:

The boing sounds each time you switch cards, or at least it does most of the time.

Depending on how your stack is set up, you may run into some exceptions. Just to make sure you do, let's continue with the example.

4. Create a new card called `No Help` and open its script.
5. Enter the following command handler in the script:

```
on openCard
  visual effect checkerboard to black
  go to this card
end openCard
```

6. Leave the card and return to it.

Result:

You get the checkerboard visual effect, but no boing.

The problem here is that the HyperCard hierarchy causes the `openCard` message to be sent to the card level first. If there is no openCard handler there, the message continues on to the background and then to the stack, where it finds the openCard handler. This last card, however, has an openCard handler, so the message stops there.

Fortunately, HyperTalk provides a method for letting a message be handled on more than one level: the `pass` command (see chapter 6, "Messages"). You can use the `pass` command in a handler to pass a message on to the next hierarchy level when the current handler is finished. To refresh your memory, Figure 14-15 shows its syntax.

```
pass <message-name>
```

Figure 14-15. The `pass` command syntax.

Try it.

Example 16:

1. Return to your HyperTalk Bible stack and the No Help card.
2. Open the card script and modify it to read as follows:

```
on openCard
  visual effect checkerboard to black
  go to this card
  pass openCard -- new command passes message one
end openCard
```

3. Return to the card level, go to another card, and return.

Result:

When you return to No Help, you get the visual effect, then the boing.

The `pass` command passes the message as soon as it's encountered in a handler. If, for example, you add a `beep` command after the `pass` command line in Example 17, the beep is skipped.

Once you see how this example lets you get a boing out of all your cards, you may choose to remove it from your HyperTalk Bible stack in order to preserve a saner working environment.

Modifying Existing Commands

You can't really change a built-in command itself, but, as you saw in chapter 6, "Messages," with `pass`, you can alter what happens when you give a command.

Chapter 6 was quite a few chapters ago, so let's review this topic. What makes command redefinition possible is that commands also work by sending messages up through the hierarchy. Therefore, if you set up your own handler for the command message, you can intercept it before it reaches the system handler for the command. Then, you can substitute or add your own instructions and pass the message on.

For instance, suppose that you want to customize the `answer` command so that it sounds a harpsichord note when it displays the answer box. You can intercept the `answer` message with your own handler.

Example 17:

1. Go to your HyperTalk Bible stack and open the card script for No Help.
2. Enter the following handler:

```
on answer
  play "harpsichord" tempo 500 84
  pass answer
end answer
```

3. Return to the card, open the Message box, and enter

```
answer "Hear that?"
```

Result:

An answer box displays `Hear that?` and the Macintosh plays a harpsichord note.

With this example, the `answer` message is passed after the `play` command is executed.

In general, if you want to modify the behavior of a command but still have the system command do the basic task, you use the `pass` command in your handler. If you want your definition of a command to replace the system definition, omit the `pass` command from your script.

You can use parameters passed along with the command, but you can't alter them. To make them available, include a parameter list in the handler.

4. Open the No Help card script and modify the answer handler:

```
on answer
  repeat for the paramcount times
    play "harpsichord" tempo 500 84
  end repeat
  pass answer
end answer
```

5. Go to the Message box and enter

```
answer "Hear that?" with "Yes"
```

Result:

The answer box displays `Hear that?` along with a Yes box. The Macintosh plays three notes.

Why three notes? HyperTalk considers the command line to have three parameters: the string `Hear that?`, the keyword `with`, and the string `Yes`.

We said you can't alter the parameters. Suppose that you try this handler:

```
on beep count
  multiply count by 2
  pass beep
end beep
```

Then, you give the command `beep 3`. The handler creates a new variable called `count`. The script then multiplies `count by 2`, making it 6. But all you hear are two beeps; changing count does not change the parameter list passed on up the hierarchy.

Modifying Function Behavior

You can use the same technique with functions as with commands. However, because the technique doesn't let you change the values of parameters, it's not that useful. For example, you can place the following handler into your stack:

```
function sin x
  answer "Remember:" && x && "should be in radians"
  pass sin
end sin
```

Because altering `x` in this handler has no effect on the parameter passed on to the system `sin()` function, this is not a big help. If you could change the parameter, you could then convert the `x` to degrees.

Suppose that you want to be able to use degrees. One approach is to write a degrees-to-radians conversion function. A second is to define your own function. Let's see how that works.

Example 18:

1. Go to your HyperTalk Bible stack and open the stack script.
2. Add the following scripts:

```
function sin_d x -- uses degrees instead of radians
  return sin (x * pi / 180)
end sin_d

function cos_d x -- uses degrees instead of radians
  return cos (x * pi / 180)
end cos_d

function tan_d x
  return tan (x* pi /180)
end tan_d
```

3. Enter

```
sin_d(30)
```

in the Message box.

Result:

The Message box returns `0.5`, the sine of 30 degrees.

Now, you can use `sin()` for radians and `sin_d()` for degrees.

Where Has Your Message Been? (Using *the target*)

The `pass` command sends a message further up the hierarchy. Sometimes, though, you need to look in the opposite direction and ask at which object the message first entered the hierarchy. This object is called the *entry point,* and HyperTalk uses `the target` to represent the name of the entry point. It has the syntax shown in Figure 14-16.

```
the target
target()
```

Figure 14-16. The `the target` function syntax.

Go back to Example 16. Suppose that you want to modify the boing so that each card gets its own distinct pitch. You can, for instance, add the card number to a base frequency. To do that, you need to know which card was opened. If the openCard handler is in the card script, there's no problem; you can refer to this card. However, your script is in the stack, so how can the script know which particular card was just opened? The answer is to use the function `the target`. It returns a string that identifies the object that was the entry point. It does so by giving the object type and its ID, as in card ID 3290. You can use HyperTalk functions to convert this information to other forms. For instance, if the entry point is a card, `the number of the target` yields the card number, which is its position in the deck. Try this approach.

Example 19:

1. Go to the HyperTalk Bible stack and open the stack script.
2. Modify the openCard handler to read as follows:

```
on openCard
  play boing  59 + the number of the target
end openCard
```

3. Return to the card level and move through several cards.

Result:

Each card produces its own frequency of boing when you switch to it, with the exception of those cards having their own openCard handlers and no `pass openCard` command.

the result Function

Some commands work slightly differently in HyperTalk from the way they do in HyperCard. Suppose, for example, that your HyperTalk Bible stack does not contain a card named Snibbleflort. (If, for some reason, you do have a card by that name, substitute another name in the following example.)

Example 20:

1. Go to the Custom Card of your HyperTalk Bible stack and open the Message box.
2. Enter in the Message box

```
go to card "Snibbleflort"
```

Result:

Your Macintosh beeps, and an answer box tells you that there is `No such card`. This is the normal behavior you've come to expect from HyperCard.

3. Cancel the answer box and create a button called `Go Card`.
4. Give the button the following script:

```
on mouseUp -- Go Card button
  ask "Go to which card?"
  if it is empty then exit mouseUp
  else go to card it
end mouseUp
```

5. Click the button. A dialog box asks `Go to which card?` Respond with `Snibbleflort`.

Result:

Nothing happens: no beeps, no complaints, no action. Nothing tells you that the card does not exist.

When, in a script, you use a `go` command to take you to a nonexistent destination, you are not informed of the problem. The same is true if you use `find` in a script to search for a nonexistent string. You can keep informed by using the `result` function. If a `go` or `find` command in a script is successful, the result returns the HyperTalk constant `empty`, that is, it returns an empty string. If the `go` or `find` command fails, the `result` returns the string `Not found`. See Figure 14-7 for the syntax of the `result` command.

```
the result
result()
```

Figure 14-17. The `the result` function syntax.

Use it to fix your button.

Example 21:

1. Go to the Custom Card of your HyperTalk Bible stack and open the Go Card button.
2. Change its script to read as follows:

```
on mouseUp -- Go Card button
  ask "Go to which card?"
  if it is empty then exit mouseUp
  else go to card it
  if the result <> empty
  then answer "Gee, I can't find that card!"
end mouseUp
```

3. Now, click the button and ask for `Snibbleflort`.

Result:

The Macintosh displays an answer box stating `Gee, I can't find that card!`

This use of the function `the result` is an example of what is termed *error-checking*. Here, you check whether `go` can perform the requested action. If it can't, you are told so. Note that testing whether `the result is not empty` is simpler than testing `if it is Not found`.

Finding *the version* of HyperCard

One of the problems with any computer language is keeping track of "versions" of the software. As a manufacturer improves the product or fixes mistakes in its operation (bugs), a new version of the product must be issued to users. This is particularly a problem with software that generates documents, such as HyperCard does with its stacks.

When a stack is created with HyperCard, the stack is given the version number of its creator. If HyperCard is 1.0.1, the stack is 1.0.1. When this stack is run under a newer version of HyperCard, it still works fine (although it won't take advantage of any of the nifty new features unless you do some scripting). However, if a stack is made under a particular version of HyperCard, and then it is opened under an older version of HyperCard, you may have problems, and the stack may not run (although it can be opened). These problems occur if the features not supported by the older HyperCard are used in the newer version.

A way to avoid problems is to use the `version()` function. What this function does is allow you to determine from your script immediately after the stack is opened what version of HyperCard created the stack. Then, you can abort the stack if the version being used isn't the same as the one the stack was created with.

Here is the syntax of the version:

```
the version
```

or

```
version()
```

The function returns a number representing the version, such as 1.0.1, 1.1, 1.2, 1.2.1, 1.2.2, etc.

The following example shows how you use the `version()` function.

Example 22:

Write an openStack handler that checks the version and aborts if the HyperCard running is lower than 1.2.

```
on openStack
  if the version < 1.2 then
    answer "This stack needs HyperCard version 1.2."
    set lockMessages to TRUE
    go home
  else
    global userHold
    lock screen
    put the userLevel into userHold
    set userLevel to 5
    -- any other user code goes here
    --
    unlock screen
  end if
end openStack
```

Result:

In this openStack handler, we use an `if` statement to check the version of HyperCard that is running. The handler also gets the current user level, changes it to 5 (scripting), runs the stack, and is able to restore the user level when we execute a closeStack later.

Enhancing the Version Function

When HyperCard changed versions from 1.1 to 1.2, Apple enhanced the `version` function a great deal (or maybe they just allowed it to filter less). There are now two new forms of syntax:

```
the [long] version [of HyperCard]
the version of stackDescriptor
```

The variable `stackDescriptor` is the description of any stack available to the Macintosh as described in chapter 3, "Objects."

When you use the form `the long version of HyperCard`, a single string is returned that tells you about the same information as found in the 1.1 version of the `version`, namely, the version of HyperCard that the stack was made with.

```
0120800
```

In this case, you can see that the stack was created with version 1.2.0. (Note that the 8000 following the version number can be ignored.)

If you used `the version of stackDescriptor`, HyperCard returns a comma-separated, five-item string that tells you the complete history of the stack. Our Home stack, running under HyperCard 1.2.2, reported the following.

```
00000000,01208000,01208000,01228000,2680958257
```

00000000 is the version of HyperCard that created the stack. When this value is all zeroes, the version used to create the stack is lower than 1.2.

01208000 is the version of HyperCard last used to compact the stack.

01208000 is the version of HyperCard that changed the stack since it was last compacted.

01228000 is the version of HyperCard that last changed the stack.

2680958257 is the date when the stack was most recently modified and closed (in seconds since midnight, January 1, 1904).

Note that items 1 to 4 are all set to 00000000 if the version of HyperCard that created the stack is lower than 1.2, so, for example, an old stack running under HyperCard version 1.2.2 gives

```
00000000,00000000,00000000,00000000,2657036958
```

You can use the `convert` command, which is described in chapter 12, "Date and Time," to convert the last item from seconds to a more readable format.

Apple recommends that you upgrade to the latest version of HyperCard. For stack publishers with legitimate products to distribute, Apple makes available the latest version of HyperCard for a small license fee of $50 per year. This can be included with your stack, so the user can upgrade and run your stack. Note that other than a Home card, Apple does not provide the bulk of stacks that come with the bundled version of HyperCard.

Summary

By writing your own handlers, you can create your own functions and commands. The placement of these handlers controls the scope of your creations. If you place a handler into a card script, only the buttons, fields, and Message box in that card can use it. Placing the handler into a background makes it available to all the cards using that background. Placing the handler into the stack allows all the objects in the stack to use the handler. Finally, placing the handler into your Home stack makes it available to all your stacks.

If the same handler name appears on two or more levels, the first handler reached by a command message or function call becomes the operative one. By using the `pass` command, you can have it pass the message or function call on down the hierarchy to the next handler.

How do you decide between using a command or a function? In general, commands are used to instigate some sort of action, such as navigating to another card

or making a button disappear, whereas functions are used to provide a value, such as the square of a number or a part of a string.

The main point is that HyperCard gives you the freedom to create your own functions and commands and the flexibility to find solutions to your particular needs.

Review Questions

1. Consider this function definition:

```
function big x, y
  put abs(x) + abs(y) into z
  return z
end big
```

   a. What is the name of this function?
   b. How many parameters does this function have?
   c. What are this function's parameter(s) called?
   d. What is the correct way to call this function for values 4 and −8?

      f. `function big 4, -8`
      g. `the big of 4 and -8`
      h. `the big(4,-8)`
      i. `big 4, -8`
      j. `big (4,-8)`

   e. The return value for part c (when x is 4 and y is −8) is

      k. −4
      l. z
      m. 4
      n. 12
      o. big

2. Consider the following function

```
function veryBig x, y
  put abs(x) + abs(y) + 100 into z
  return z && "is a very big number"
end veryBig
```

   and suppose that we pass it the values 4 and −8.

   a. What is the function's return value?
   b. How could you access just the numeric part of the return value?

3. One difference between user-defined commands and functions is

   a. only commands use handlers
   b. commands can return more than one value
   c. only commands can return a value
   d. functions are more handsome than commands

4. How is the command `pass thisMessage` like `exit thisMessage`? How is it different?
5. Suppose that you place a mouseUp handler into a stack. Which of the following actions normally would activate this handler? Explain the exceptions, if any, briefly.

   a. clicking a normal field
   b. clicking a text-locked field
   c. clicking a card
   d. clicking a button

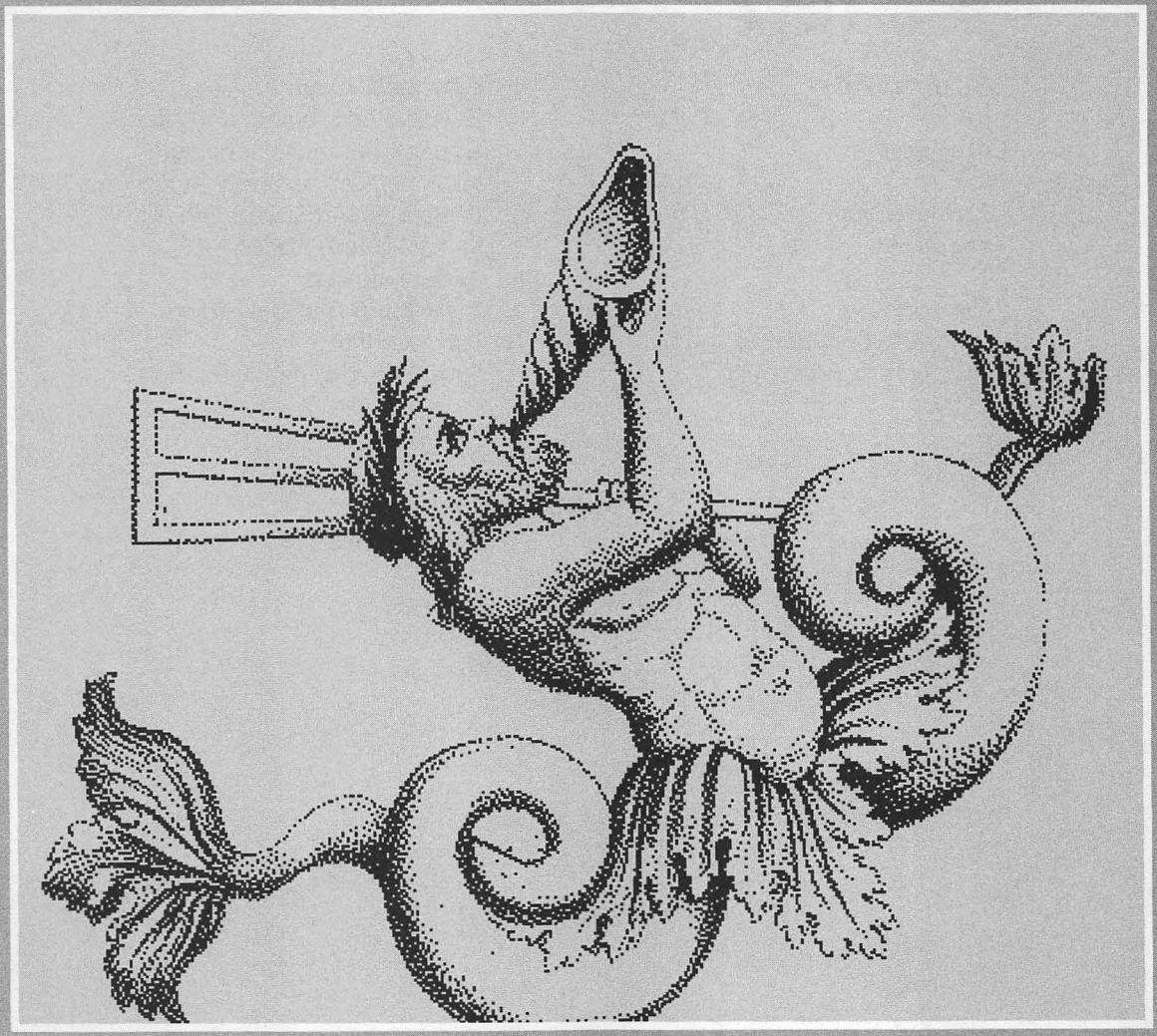
6. What does the function `the target` return?
7. What does the function `the result` return?
8. What are the two ways to access parameters passed along with a command?
9. Can both methods listed in your answer to Question 8 also be used with functions?

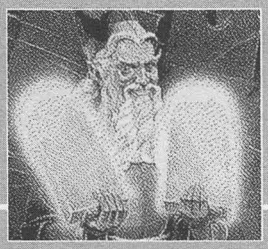
Exercises

1. Write a function that returns the difference between two numbers divided by their sum.
2. Write a function that returns characters m through n of a string. Have the function return if n < m or if m exceeds the length of the string. Set m to 1 if m < 1. If n exceeds the length of the string, go to the end of the string.
3. Write a function that takes the two short sides of a right triangle as parameters and returns a string containing the hypotenuse and the angle between the first side and the hypotenuse. Hint: See chapter 13, "Math Functions," for the necessary math.
4. Write a string function that returns the words of its argument in reverse order.
5. Write a command `dialnote` that takes two parameters: a note number and a repetition number. The command should use the `note` command to play the given note the specified number of times. Use the `param` family of functions. Have the command display an answer box if not enough parameters are given.
6. Write and place a handler that causes any button on one particular card (but not any other card) to disappear for 10 seconds when a `mouseDown` signal is sent to the button.

Chapter 15

Advanced Sound





Quick View

Commands
`play`
`play stop`

Functions
`the sound`

Concepts
Quick introduction to sound
`play` command syntax

Concepts *cont'd*
Harpsichord, boing, and silence
HyperTalk's music language
Note, octave, duration, and tempo
Project script: Sound Lab
Special sound effects
`Sweep` sounds
High-Tempo special effects
Jazz handler
Sound-Card synchronization

Chapter 15

Advanced Sound

In this chapter, we explore the outer limits of making sound and music using HyperTalk's `play` command, a sound generation command that implements a powerful set of musical notations and is pretty much a complete language in itself. We don't spend too much time on the theory of music—there are plenty of other books that do that.

We also build a sound experimenter's laboratory called Sound Lab to help you understand how the `play` language works. The Sound Lab is an easy-to-use-interface to HyperTalk's `play` statement that lets you explore HyperCard's built-in voices and make special sound effects. You also learn how to use a mouse click and the `play stop` command to stop a sound from playing, and how to synchronize sound with screen graphics.

A Quick Introduction to Sound

Before we get too wrapped up in the mechanics of HyperTalk, let's review the nature of sound waves and how they are described. That makes the subsequent HyperTalk terminology much clearer. (For more information, consult *The Science of Musical Sound* by John R. Pierce, published by Scientific America Library in 1983.)

Sound, at its most basic level, is the result of changing pressure waves striking the ear drum. When these waves are pleasant to humans (and maybe other species—so far none has stepped forward), the sound is called music. These waves are generated by some object moving the air—a person's vocal chords, an automobile's exhaust, a rock hitting the ground, your dog barking, etc. Sound is made up of pressure waves and these waves have certain characteristics that can be identified.

Among all the fluctuations of pressure are the musical sounds. These occur when the air pressure rises and falls periodically with time. If you attach a microphone to a cathode-ray oscilloscope, the sound pressure waveform is traced. The

most revealing waveform to examine is the one that is produced by a "pure" tone, or *sine wave*, in which the air pressure rises and falls sinusoidally with time.

Sine waves sound pretty boring and are rarely found in nature, yet they contain much information about music and are important to understand. Recall that HyperTalk has a function called `sin`. This function can be used to demonstrate graphically all the characteristics of a pure tone, so let's write a simple script to generate a sine wave. Note, in the next two examples, we use the HyperTalk Bible stack.

Example 1:

1. Create a new card at the end of your HyperTalk Bible stack and call it `Sine`. On this card, make a new button, position as shown in Figure 15-1, name the button `Sine wave`, and enter the following handler:

```
on mouseUp -- sine pattern generator
  doMenu "Background"
  choose brush tool
  set brush to 32          -- make 4-pixel-square drawing pen
  doMenu "Select All"      -- clear the background screen
  doMenu "Clear Picture"
  put 4 * pi into a2       -- upper limit of angle in radians
  put 100 into r           -- set the amplitude
  put a2 / 180 into inc    -- size of each increment in radians
  put 512/a2 into factor -- squish horz by decreasing a2
  put zero into angle
  repeat for 180 times
    put trunc(angle * factor) into x
    put trunc(150 - r * sin(angle) into y
    put "x = " & x && "y = " & y into msg
    click at x,y           -- actual brush drawn here
    put angle + inc into angle -- increment angle
  end repeat
  choose browse tool
end mouseUp
```

   You don't have to enter the labels shown in Figure 15-1; they are only for the purpose of discussion.

2. Click the Sine wave button.

Result:

A wave is drawn with two peaks as shown in Figure 15-1, sans labels and arrows.

You should understand how this handler works by now, because we have covered all the topics that comprise it. A sine wave is made up of three basic components: amplitude, period, and phase. We can ignore phase for this discussion.

The *amplitude* of a sine wave is the amount of air pressure it generates when the air is moving, and is represented by the height of the wave (a) above and below (−a) its average. A strong wave is the result of a large pressure and is loud. A small wave is the result of low pressure and is faint. The ear can detect an amazingly large

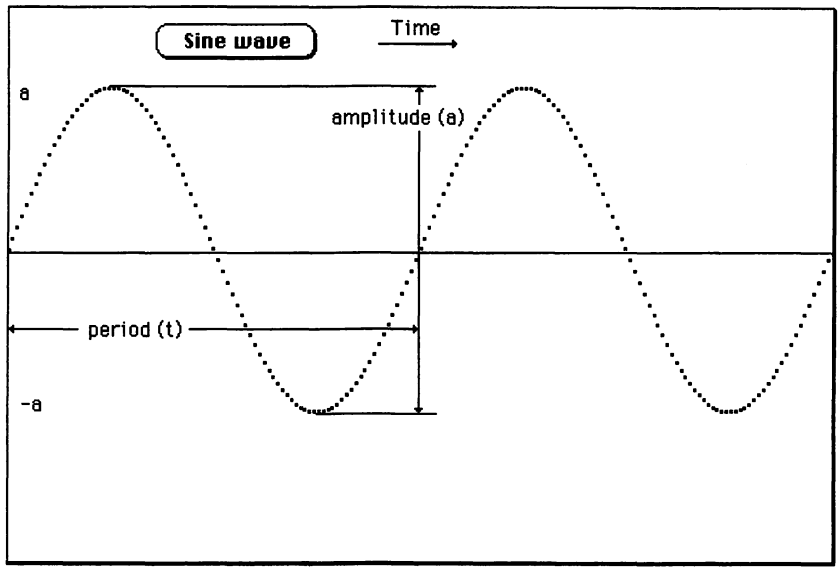


Figure 15-1. Sine wave drawn in HyperCard.

range of differences in air pressure, and the power it takes to move this air is usually used to describe the volume of a sound. The Mac's tiny two-inch speaker only has two-tenths of a watt driving it and can't move much air, so you can't hear sounds very well with it. For this chapter, run the output of your Mac's speaker to a hi-fi or an external amplifier.

The *period* of a sine wave is the amount of time it takes to go from one peak to the next (t). The reciprocal of t is 1/t, and therefore gives the number of peaks per second, which is the frequency (f) of the sine wave (f = 1/t). The higher the frequency of a waveform, the higher the pitch must be, and the shorter the time between peaks. Frequency is usually measured in cycles per second, or cycles. Today, the term Hertz, honoring the physicist Heinrich Hertz (who had nothing to do with rental cars) is used to represent cycles per second. The abbreviation of Hertz is Hz.

The human ear can hear an average of 20 to 20,000 cycles per second. Most speech takes place around 300 Hz, the piano goes from 27.5 Hz to 4186.0 Hz, and wind and ocean waves comprise the higher frequencies of around 10,000 Hz.

Piano notes are not pure sinusoids as in our figure, but are, rather, made up of a pure sine wave called the *fundamental frequency*, and a large number of multiples, or *harmonics*, of the fundamental. These harmonics are added into the pure sine wave, resulting in a rich-sounding note. The number of harmonics and the upper frequencies that they add to the fundamental are what give each note its characteristic sound.

These two characteristics, amplitude and frequency, are all that we need to explain how to use HyperTalk to make music.

The *play* Command Syntax

Now that we know some waveform theory, let's review the `play` command in more depth than it was covered in chapter 3, Audio and Visual Effects." The syntax for the `play` statement is illustrated in Figure 15-2.

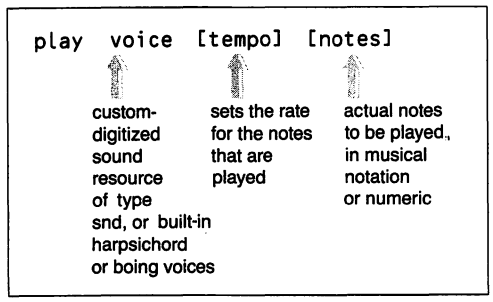


Figure 15-2. The `play` command syntax.

For a reminder, here is a simple example of `play` that can be typed into the Message box:

Example 2:

Enter

```
play "harpsichord" tempo 200 "c d e f g a b"
```

in the Message box and press the Return key.

Result:

You hear the notes of the western musical C-major scale (notes C through B) played in the harpsichord voice.

The *voice* and *tempo* parameters

The `play` statement is always followed by a `voice` string. The `voice` string, which may be a container, is the name of a sound effect that has been sampled digitally and stored in HyperCard. There are three built-in voices in HyperCard: harpsichord, which sounds like a 17th-century piano, boing which sounds like a spring popping open, and silence, which, not surprisingly, sounds like nothing. For this section, we use the harpsichord sound effect. Later you will see how to add your own sounds to HyperTalk.

The `tempo` parameter controls how fast the notes are played. When not used, the default is 120 beats per minute. In our example, `tempo` has been increased to 200 beats per minute. This parameter may also be a container.

HyperTalk's Music Language

The final parameter in the `play` statement is the `note` string. The `note` string is made up of notes, octave numbers, durations and other characters that comprise a complete built-in musical language. The language is based on the western (occidental) music scale. Figure 15-3 shows the syntax for this language. This example shows the note is C-sharp `c#`, is in the octave below middle `3`, is an eighth-note `e`, and is dotted (.) to indicate that it is to be held longer.

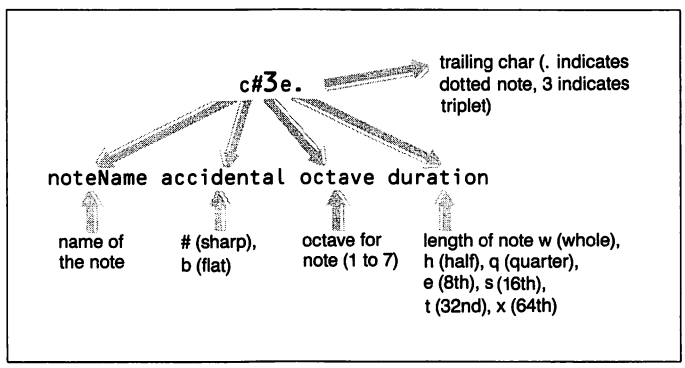


Figure 15-3. The syntax for the notes used with the `play` statement.

The language is composed of notes representing the notes of a piano keyboard. Each note forms a distinct frequency in a pattern that western musicians have agreed to follow over the millennia—12 notes in an *octave*. An octave higher doubles a particular frequency; an octave lower halves a particular frequency.

If everyone agrees that the note middle A on the scale is at 440 Hz, you can create the entire rest of a musical scale from it. Simply find each octave. There is one at 880 Hz, 1760 Hz, 3520 Hz, 7040 Hz, and 14,080 Hz. Lower octaves are located at 220 Hz, 110 Hz, 55.5 Hz, and 27.5 Hz. Divide the frequency range between each octave into 12 even ratios, so that each note is actually the twelfth root of 2 ($2^{1/12}$) higher than the note before it.

A scale with these mathematical ratios is called the *equally tempered musical scale*. Keep in mind that the frequency differences between subsequent notes are not actually equal, but the ratio of the notes *is* equal. These notes and the corresponding frequencies for the entire piano keyboard are shown in Figure 15-4.

Table 15-1 shows more HyperTalk details on the notes of the fourth octave of the piano keyboard, the one starting at C = 261.63 Hz (called middle C on the keyboard). As you can see, the black keys on the piano keyboard are named as special

cases of the keys on either side. You learn to use these later in the Sound Lab. The C-sharp note on the keyboard is the same as the D-flat note on the keyboard. HyperTalk uses the notation # to stand for sharp and b to stand for flat.

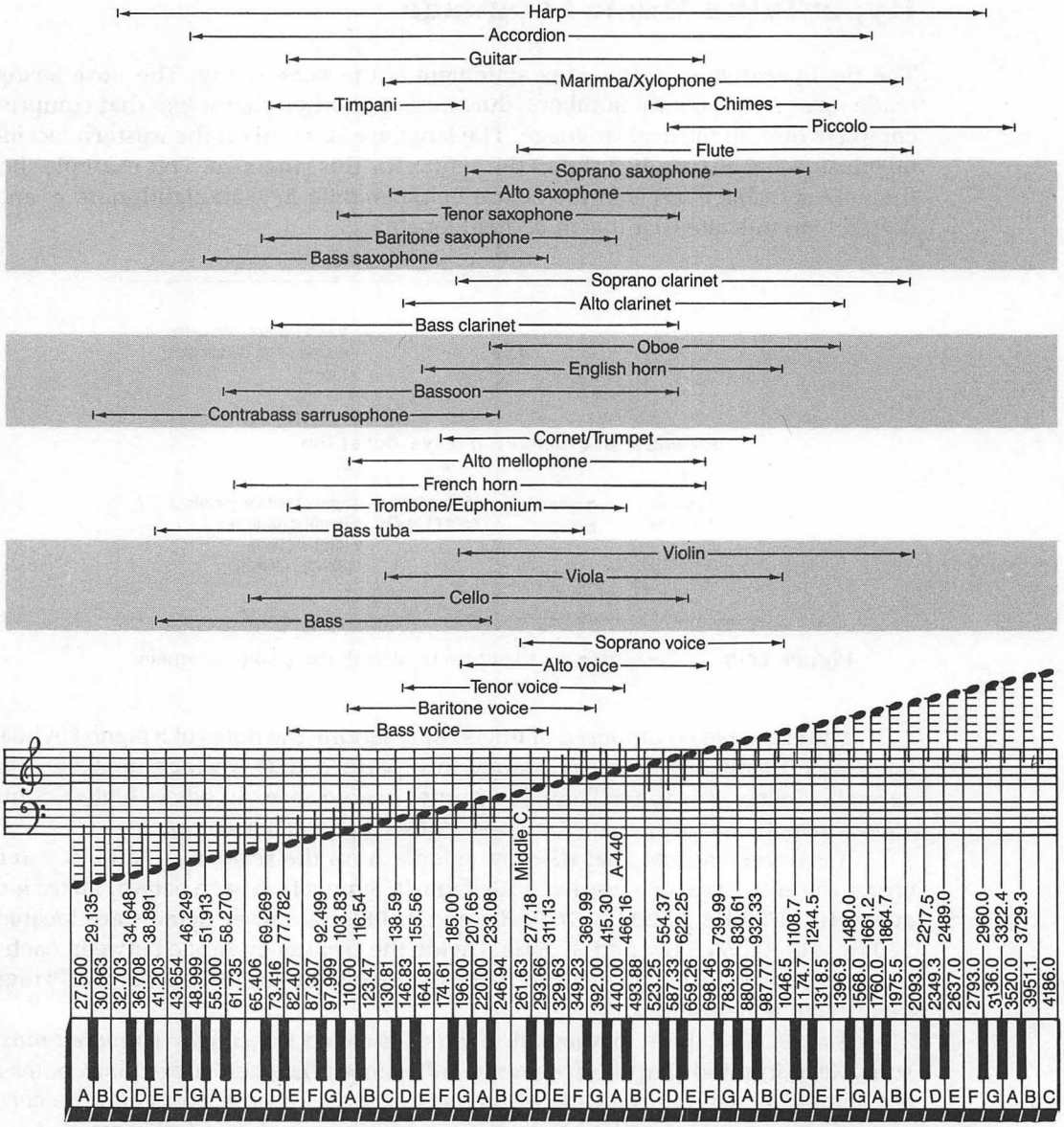


Figure 15-4. The piano keyboard and the frequencies corresponding to each note.

Table 15-1. The Fourth Octave and Its HyperTalk Notation

Note No.	Sharp	Flat	Frequency	HyperTalk	Key Color
60	C	C	261.63	c	white
61	C-sharp	D-flat	277.18	c#, db	black
62	D	D	293.66	d	white
63	D-sharp	E-flat	311.13	d#, eb	black
64	E	E	329.63	e	white
65	F	F	349.23	f	white
66	F-sharp	G-flat	369.99	f#, gb	black
67	G	G	392.00	g	white
68	G-sharp	A-flat	415.30	g#, ab	black
69	A	A	440	a	white
70	A-sharp	B-flat	446.16	a#, bb	black
71	B	B	495	b	white

In Figure 15-3, there are four possible sets of characters: `noteName`, `accidental`, `octave`, and `duration`. Let's examine each of these parameters and learn how each works.

Extending HyperTalk's Musical Scale

Other musical scales are built on different ratios between the notes. For example, some Eastern scales have five or six notes. Creating such scales on HyperCard would be terrific, but unfortunately, there is no way to do that. Perhaps the founding fathers will find a way to create a sound statement like IBM's BASICA that accepts an array of user-defined musical scales stored as resources, or perhaps as strings.

Note Syntax: Accidentals

The *note name* (`noteName`) is one of the standard notations allowed in Table 15-1. The *accidental* is the sharp (`#`) or flat (`b`) and, if used, immediately follows the note name. The table shows the equivalency between notes using the sharp and flat nomenclature. In the Western scale, C-sharp and D-flat are two ways of saying the same thing—each of the black keys has an alias and can be specified either as a sharp of the note below it, or as a flat of the note above it.

Example 3:

1. Go to your HyperTalk Bible stack, create a new card at the end of the stack, name it `Practice`, create a new button, name it `Play.Sharp.Flat`, and arrange as shown in Figure 15-5.
2. Create two new wide horizontal fields, name one `x` and the other `y`, and arrange as shown in Figure 15-5. Make the font in each field 9-point Monaco. We use the fields so you can look at the notes as you hear the sound.
3. Enter the following handler in the button:

```
on mouseUp -- Play.Sharp.Flat button
  put "c c# d d# e f f# g g# a a# b" into card field x
  put "c db d eb e f gb g ab a bb b" into card field y
  play "harpsichord" tempo 200 card field x
  play "harpsichord" tempo 200 card field y
end mouseUp
```

4. Click the button Play.Sharp.Flat.

Result:

The two sets of notes appear in the respective fields and the `play` statements execute them one after another. They sound exactly alike.

The two strings contain the equivalent ways to express the same basic scale. Keep in mind that you can't allow your `note string` to break across a line (using option-return) without repeating the `play` command.

Note Numbers

HyperTalk also allows you to specify the notes by number rather than letter and accidental. Table 15-1 and Figure 15-4 show the note numbers corresponding to the note names. The best way to remember this relationship is that middle C on the keyboard is note number 60. Thus, the note represented by 61 is C-sharp and 59 represents C-flat. You can use the numbers in lieu of the `noteName` parameter when you wish to generate the notes mathematically, as you see in subsequent examples.

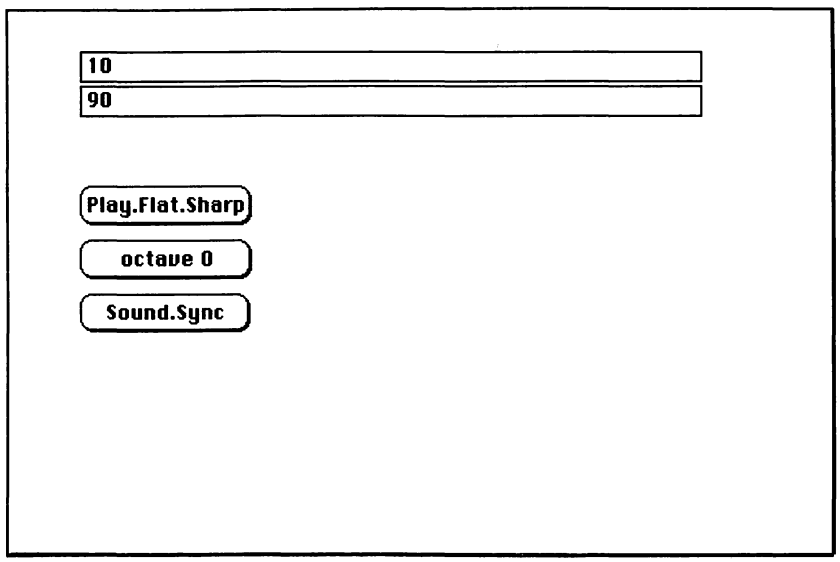


Figure 15-5. The `play` statement practice card.

Octave Number

The octave number follows the accidental character. It controls the overall area of the keyboard where the notes are played. The highest note is B7 (3951.1 Hz), and the lowest is C0 (16.352 Hz). It is interesting that the piano keyboard doesn't go down to C0, but, rather, goes down only to A1, which is 27.5 Hz, as shown in Figure 15-2.

If you don't type an octave number, the default is the fourth octave which starts with middle C. Keep in mind that the octave applies to all subsequent notes or until the next octave number occurs.

Example 4:

1. Create a new button on your card Practice, name it `octave`, and enter the following handler:

```
on mouseUp
  global octave
  put (octave + 1) mod 8 into octave -- 0 to 7
  play "harpsichord" "b" & octave
  set the name of the target to "octave" && octave
end mouseUp
```

2. Click the button once.

Result:

The button's name changes to a new number and plays the note B in an octave. Each successive click increments the number of the button by 1 and plays the note B in the next octave, so you hear B0 through B7.

This shows the impressive range of the Macintosh and HyperCard sound generation software. When you try this script, be aware that the sound at the lower octaves is not very loud, especially on a Mac Plus.

The script for the button is a trick. The statement

```
put (octave + 1) mod 8 into octave
```

increments the variable `octave` each time the button is clicked, and uses `mod` to restrict the range of `octave` from 0 to 7. You must use global `octave` in the handler; otherwise, HyperTalk can't execute the previous `put` statement. The statement

```
set the name of the target to "octave" && octave
```

sets the name of the button to the text `"octave"` followed by the number 0 to 7. The `&&` ensures there is a space between the word `octave` and the number.

Duration

The *duration* is the length of a note compared to the other notes. There are seven possible durations, as shown in Table 15-2 and Figure 15-3. A whole-note is the longest note. (When using whole-notes, be careful not to set the `tempo` parameter too

high or you won't hear them properly. We explore more about tempo soon.) Half-notes are one-half the length in time of whole-notes, quarter-notes are one-half the length of half-notes, and so on, down to 64th-notes.

Table 15-2. Note Durations in the `play` Command

Symbol	Duration
w	whole-note
h	half-note
q	quarter-note (default)
e	eighth-note
s	16th-note
t	32nd-note
x	64th-note

Once a duration has been specified, it remains in effect until the duration is changed again. Let's learn what these durations sound like.

Example 5:

1. Enter `play "boing" "c c c ab3"` in the Message box and press Return.

Result:

HyperCard plays the first four notes of Beethoven's Fifth Symphony.
The first three notes are C, using the default octave (the fourth octave). The duration for the first three notes is the default of quarter-notes. The last note played is A-flat (`ab`) in the third octave below middle C. The default tempo is 120 beats per minute (2 beats per second = one beat each half-second). The timing of the notes sounds odd.

2. Insert your cursor after the note `c` and type the letter `e`; Insert your cursor after the 3 in the last note and type a `w` so that the Message box contains `play "boing""ce c c ab3w"`. Press Return.

Result:

HyperCard plays the first three notes as eighth-notes.
This means that the duration of the notes is one-half the previous quarter-notes. The script then plays the last note as a whole-note, which extends it to eight times longer than the previous three C notes. The first four notes sound much more like the real thing.

3. Enter the string `bbe bb bb gw` after the fourth note so your Message box contains

```
play "boing" "ce c c ab3w bbe bb bb gw"
```

and press Return.

Result:

The second bar of music is played using three B-flats followed by a G.
Note, you had to switch the first B-flat to an eighth-note (`bbe`), because the previous note was a whole-note (`ab3w`).

The duration symbol may be followed by a period (.) character to make the note a *dotted note*. In music, a dotted note increases the duration of a note by 50 percent. In HyperTalk, a dotted note increases the duration of the note, and all that follow it, by 50 percent. This stretches the note. If you dot a quarter-note, you get a quarter-note plus half a quarter-note (an eighth-note), or a 3/8-note. If you dot a half-note, you get a 3/4-note.

The duration symbol may also be followed by a 3, which indicates a *triplet*. In music, a triplet speeds up the tempo of a particular set of notes, making the notes faster. In HyperTalk, indicating a triplet speeds up a note as well as the notes following it. It causes the three notes to be compressed into the time that two of them would take.

If you have two half-notes and set the first as a triplet, the two half-notes that follow will be included as part of that triplet and three half-notes will now fit in the space that two of them once took. A half-note that is followed by a 3 (`h3`) means that the half-notes are compressed so that three of them fit in the period that two normally take. Thus, the triplet notes appear to play faster than the normal notes. The triplet holds until another duration parameter occurs in the play string.

4. Add 3 to the end of the first note in the Message box so it looks like this:

   ```
   play "boing" "ce3 c c ab3w bbe bb bb gw"
   ```

   and press Return.

Result:

The first four notes are played faster than in the previous example, while the second four notes are played at the normal eighth-note rate. The triplet compresses the notes.

5. Replace the 3 in the first note with a dot in the Message box, so it looks like this:

   ```
   play "boing" "ce. c c ab3w bbe bb bb gw"
   ```

   and press Return.

Result:

The first four notes are played about twice as slow as when the dot isn't used, and the second four notes are played in normal tempo. The first four notes are stretched 50 percent.

Use the duration symbols to convert sheet music to codes suitable for HyperTalk. Keep in mind that after you change the duration, it remains in effect until you change it again or reach the end of the `play` statement.

The *tempo* Parameter

The `tempo` parameter controls the number of beats per minute at which the notes of the `play` command are played back. The default is 120 beats per minute, about 2 per second.

Example 6:

Enter

```
play "boing" tempo 240 "ce c c ab3w bbe bb bb gw"
```

in the Message box and press Return.

Result:

You hear the bars of Beethoven's Fifth played at twice the normal speed. The tempo is doubled from its default of 120 to 240.

What are the ranges of `tempo`? If you're writing music using HyperTalk, you do not usually need to make the tempo slower than *largo* (42 to 66 beats per minute) or faster than *presto* (between 69 and 152 beats per minute, depending on the duration of the note). However, you may want to play with ranges of tempo if you are investigating special music effects, as we see later.

The Sound of *silence*

Remember, one of HyperTalk's voices is silence. Silence is most effectively used for musical rests; its duration is also determined by the symbols in Table 15-2. The statement

```
play "silence" h
```

plays a half-note rest.

PLAY* vs. *play

BASIC hackers beware. The equivalent statement for playing music in BASIC is `PLAY`, which is a well-known musical language developed by Microsoft and standardized on the IBM PC within BASICA and dozens of other clones. The `PLAY` statement is more difficult to use in BASIC, and it offers less range. `PLAY` has a tempo range of 32 to 255 beats per minute, whereas Hypertalk's `play` can go from 1 to 32,767 beats per minute! HyperTalk's `play` runs in the background automatically, while BASICA's `PLAY` must be forced to do that. `PLAY` has execute strings that let you call another sound—sort of like subroutines. HyperTalk's `play` lacks these. `PLAY` gives a range of notes similar to `play`, but the quality of the music is poorer. One reason for this is that the quality of the sound circuitry hardware on the IBM PC is inferior to the Macintosh Plus, SE, and Mac II. The mounting and quality of the speaker on the IBM PC is poor. (Most PC speakers are mounted on the back of the computer almost as an afterthought.) One feature `play` has going for it over `PLAY` is that its notation is a lot simpler. Here is a bar from `PLAY`:

```
100 A$="cecececece-ce-ce-ce-dg-dg-dg-dg-dfdfdfdf"
180 PLAY "MFMNL16T12004XA$;03XA$;02XA4;XA4"
```

You figure it out.

Project Script: The Sound Lab

We can investigate the `play` statement by typing it into the Message box with different parameters, and following it with a string representing the notes, durations, and octaves. This is not very useful when you want to really experiment with the `play` statement. You have to memorize the seven codes for the durations, plus the dot (.) and the 3 for triplets, the # and b for sharp and flat, and also be careful not to erase your entire statement accidentally when you type into the Message box. Typing everything into custom buttons is not perfect either, since you can't see the script as you listen to it. Clearly what is needed is a simple laboratory that allows you to point and click different notes and parameters and have them automatically sent to the `play` command.

Figure 15-6 shows Sound Lab, our experimenter's work bench or simple sound laboratory. To use it, type your string of notes into the field at the top of the screen. Notes are stored in the field while you manipulate them with the various buttons that control the duration, octave, and tempo of the string. Some clever tricks using buttons and fields are involved in the Sound Lab.

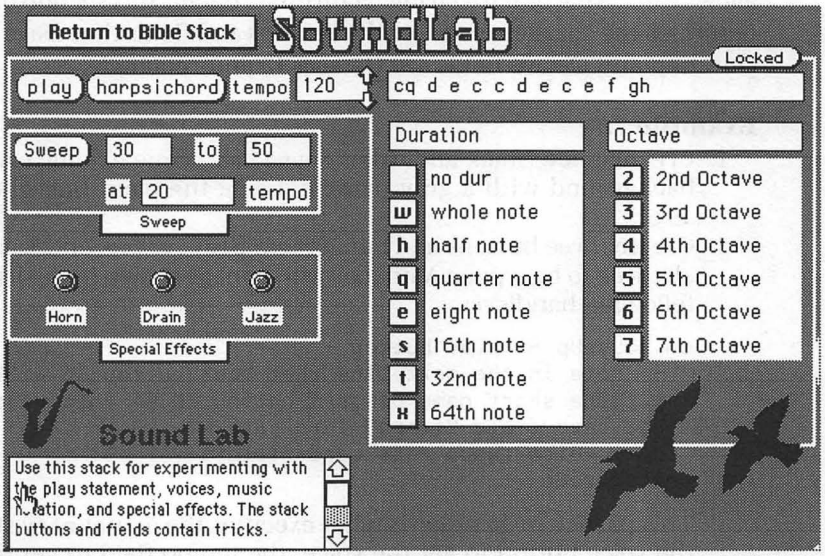


Figure 15-6. Sound Lab.

One neat feature of Sound Lab is that when you click a note in the play field at the top of the screen, Sound Lab automatically selects it and turns it black (using a

trick involving `the selection`). When you click any of the duration, octave, or tempo buttons, the note and its associated duration and octave numbers (we call it the *note chunk*) are modified by whatever button you pressed.

In Figure 15-6, if you click the first note chunk, `cq`, it immediately turns black. If you then click button 5 under `Octave`, the chunk will be replaced by `cq5`, showing you have specified the fifth octave. No typing was necessary. Your resulting string is exactly what would be typed into the `play` statement.

When you click the `play` button in the figure at the top right of the screen, you hear the string in the play field in the fifth octave. To change the note durations from quarter-notes to eighth-notes, click the middle D note in the field, and click the e button. The note now reappears in the string as `de`, specifying this is a D note, and that it and subsequent notes are to be played as eighth-notes. HyperTalk puts the new duration and octave into effect until it is changed to a new duration or octave.

To make another change in a note chunk, go back and select it again. When you select a note and then press a button, the button handler causes a letter or number to be pasted into the selected note. After that, the selected area disappears so you have to click again to make a new change.

Building the Sound Lab

Let's put this application together. We first build the section of the Sound Lab that allows exercising of `note` string. Figure 15-7 details this section of the Sound Lab and describes the characteristics of the buttons and fields that make up this section. The Duration and Octave fields are Geneva 12 and rectangle.

Example 7:

1. Create a new stack and call it `Sound Lab`. Name its only card `Sound`, and fill its background with a gray pattern, using the paint bucket as shown in Figure 15-6.
2. Create three buttons and name them `play`, `voice`, and `Locked`. Set the font for all three to be 9-point Monaco, size them as shown in Figure 15-7, and enter the following handlers:

```
on mouseUp -- play button
  -- note in the next line that your button ID will be different
  put the short name of card button ID 64 into voice
  put card field "tempo" into rate
  play voice tempo rate card field x
end mouseUp
```

This particular handler is what executes the actual `play` statement. It looks in the voice button to get the voice, the tempo field to get the tempo, and card field x to get the notes. It is unusual in that it gets the voice from the name of the voice button. The names in the voice button toggle when the button is pressed. This approach means we can't refer to the button by name, and that is the reason for the ID number in the script. Your ID numbers will be different, so you have to hand type the actual value for the voice button in the script.

Notice that the button, and several of the subsequent buttons and fields, have their text font and name size in 9-point Monaco instead of the default 12-point Chicago. This allows the buttons to be smaller.

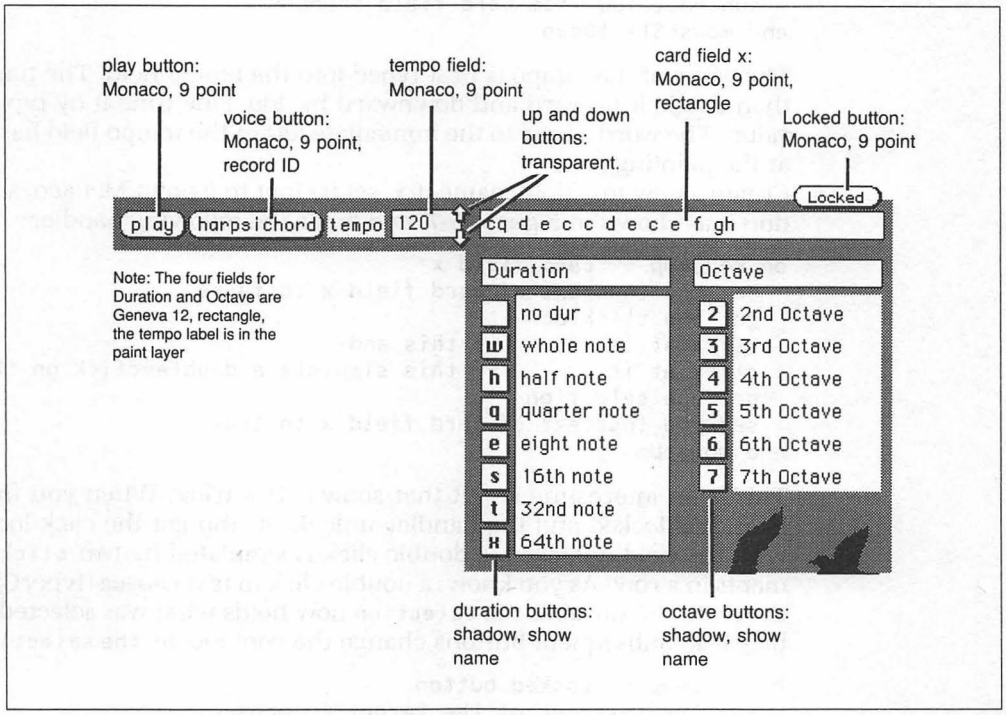


Figure 15-7. The `play note` section of the Sound Lab.

```
on mouseUp -- voice button
  global rate
  put (1 + rate) mod 2 into rate
  if rate = 0 then set the name of the target to "harpsichord"
  if rate = 1 then set the name of the target to "boing"
  play "harpsichord" 70
end mouseUp
```

The voice button allows us to toggle between different voices. This script shows how it's possible to get a button's name to change on the screen each time the button is clicked. You could use `on mouseStillDown` to make it continuously toggle upward.

3. Create a new field, name it `tempo`, make it a rectangle style, and change its font to 12-point Monaco. Position it as shown in Figure 15-7. No script is entered in this field.
4. Create two new buttons, use the up and down arrow icons, disable `show name`, and enter the following handlers:

```
on mouseStillDown -- tempo up button
```

continued

```
  add 100 to card field "tempo"
end mouseStillDown


on mouseStillDown -- tempo down button
  subtract 100 from card field "tempo"
end mouseStillDown
```

The value of the tempo is first typed into the tempo field. The tempo buttons then toggle it upward and downward by 100. Fine tune it by typing an exact value. The word `tempo` to the immediate left of the tempo field has been typed at the painting layer.

5. Create a new text field, name it `x`, set its font to 9-point Monaco, size and position it as shown in Figure 15-7, and enter the following handler:

```
on mouseUp -- card field x
  set the locktext of card field x to false
  get the clickloc
  click at it        -- this and
  click at it        -- this simulate a double-click on the word
  get the selection
  set the locktext of card field x to true
end mouseUp
```

This is an interesting script that shows off a trick. When you first click the field, it is locked and the handler unlocks it. You get the click location, then, with the field unlocked, a double-click is simulated by two `click at it` statements in a row. As you know, a double-click in text causes HyperCard to select the text. The variable `the selection` now holds what was selected by the double-click. Subsequent buttons change the contents of `the selection`.

```
on mouseUp -- Locked button
  set the textfont of the target to monaco
  set the textsize of the target to 9
  if the short name of the target = "Locked" then
    set the name of the target to "Unlocked"
    set the locktext of card field x to false
  else
    set the name of the target to "Locked"
    set the locktext of card field x to true
  end if
end mouseUp
```

The feature `the selection` does not work unless the text field is locked. This button makes it easy to lock and unlock the field (you must unlock it to enter a string) without having to open and close the button's dialog box.

At this point, it is a good idea to test your Sound Lab. Enter some notes that you know are legal into card field `x`. Make sure it is unlocked by clicking the Locked button. It should display `Unlocked`. Click the voice button until it displays `harpsichord`, and enter 120 into the tempo field. Click the play button and hear your tune played.

Try different notes. Click the tempo up and down buttons to change the tempo, and click play to hear the notes. Try changing the voice to boing by clicking the voice button.

6. Create a single card button for the note durations, delete its name, and enter the following handler. Copy and paste six new replicas of this button, line them up as in Figure 15-7, open each and change its name, as shown, from no name (blank) to the name x.

```
on mouseUp -- card buttons blank, w, h, q, e, s, t, x
  put "w h q e s t x" into array
  put the selection into unit
  if the selection = empty then exit mouseUp
  if the selection = "e" then -- must be single e note
    put unit & the short name of the target into the selection
    play "harpsichord" 60
    exit mouseUp
  end if
  if array contains last char of unit then
    play "harpsichord" 60
    delete last char of unit
  end if
  put unit & the short name of the target into the selection
end mouseUp
```

Note that this handler checks whether `the selection` is empty before proceeding. This makes sure you can't accidentally type in another character while the insertion cursor is in card field x, although nothing is really selected. The handler looks at the last character of the selection for the duration character and, if it is in the array of acceptable characters, the handler changes it. The handler must also check the one special case where the last character is also the only character and is an `e`. This could be confused with the e for eighth-note, so a special `if-then` statement hunts for this lone exception and handles it.

7. Create a single card button for the octave choices, name the first `2`, and enter the following handler. Copy and paste new replicas of this button, line them up as in Figure 15-7, open each, and change each name as shown:

```
on mouseUp -- octave buttons 2, 3, 4, 5, 6, 7, 8
  put "2 3 4 5 6 7 8" into array
  put the selection into unit
  if unit is empty then exit mouseUp
  if second char of unit is in array then
    play "harpsichord" 60
    delete second char of unit
  end if
  put the short name of the target after first char of unit
  put unit into the selection
end mouseUp
```

Using the Sound Lab

Now that you have built the Sound Lab, use it to experiment. The idea is to type your notes with the play field unlocked. Then lock the field and press the play button to hear the notes. Tempo too slow? Click the tempo up and down buttons. Try chang-

ing the duration of a note: click the note in the play field, click any of the duration buttons, and click the play button to hear the effect. Do the same thing for the octave buttons. Try the upper and lower limits of the tempo and durations to get some amazing sound effects.

HyperCard Expects format2 *'snd '* Resources
HyperCard uses sounds stored in a `'snd '` resource. Note, the space following the d in `'snd '` is required. There are two formats used for the `'snd '` resource: format1 and format2. HyperCard's `play` command requires a format2 `'snd '` resource. The `'snd '` resources supplied in the Macintosh System files are in format1 so, unfortunately, HyperCard can't access them.

Special Sound Effects

At first, it might not seem that HyperTalk's `play` statement is very flexible for making nonmusical sounds. Sure, you can play notes, but you can't control the exact frequency of the sound, nor the exact volume. Because HyperTalk's `play` language only accepts notes in the range of the piano keyboard, other frequencies can't be specified. (Sounds like a good idea for an external command.) The more subtle characteristics of a sound, such as how it rises in volume at the beginning of the note (attack) and how it dies out (decay), also can't be controlled. Because the `play` statement accepts a very wide range of tempo and notes, however, we can get it to produce some rather weird effects. Let's see how to do that.

Sweep Special Effects

You can generate some unusual frequency sweeps with HyperTalk. We can modify the Sound Lab to add a frequency-sweep testing center. Refer to Figure 15-8 which shows the characteristics of the objects for the Sweep section.

Example 8:

1. Copy the play button, change its name to `Sweep`, position it as shown in Figure 15-8, and enter the following handler:

```
on mouseUp -- Sweep button handler
  put card field "rate" into rate
  put "e" into duration
  -- your ID number for the next line will be different
  put the short name of card button ID 64 into voice
  repeat with note = card field low to card field high
    play "harpsichord" tempo rate note & duration
  end repeat
end mouseUp
```

This handler simply takes the contents of the card field rate, puts it into the

variable `tempo`, sets the duration to an E note (you can change this) takes the voice out of the `play` statement's voice button (because it's already there), and enters a `repeat with` loop. The loop uses the card field called low to obtain its starting point and the high field to obtain its upper limit. Finally, inside the loop, we execute the `play` statement using the variables `rate`, `note`, and `duration`.

2. Copy the card field called tempo from the `play` section you built. Paste three copies, name the copies `low`, `high`, and `rate`, and arrange them as shown in Figure 15-8. Add the letters `to`, `at`, and `tempo` with the paint tools.

   The Sweep section is done. Let's test it.

3. In the browse mode, enter 20 in the low field, 100 in the high field, and 1000 in the rate field. Click the Sweep button.

Result:

You hear a low-frequency beat, like a humming, that grows in frequency, climbing up the scale until it hits note 100 and stops.

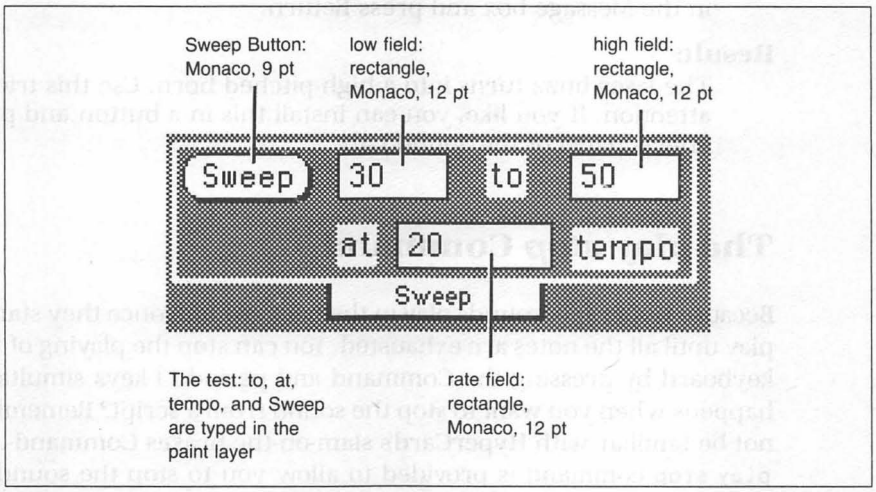


Figure 15-8. The `Sweep` section of the Sound Lab.

The Sweep button is a good way to discover the variety of frequency and tempo variations that have potential for sound effects. For example, from this script, you discovered that the notes in a sweep ranging from 20 to 50 sound exactly like the engines of a starship when they are about to blow up.

Boing Special Effects

You can find some interesting effects in the boing voice. For example, at low frequencies, boing shows some strange behavior. Try using the Sound Lab to play note number 30 with a tempo of 30. It sounds like water going down a drain. Whenever you discover a new interesting sound effect, you can put it in a button and install the button in the special effects frame.

High-Tempo Special Effects

Example 9:

1. Enter

```
play "boing" tempo 32767 "c6w c c c c c c c c c c c c c c c"
```

in the Message box and press Return.

Result:

You hear a medium-frequency buzz, rich with harmonics, that sounds like a laser blast from a George Lucas film. Sixteen C notes in the sixth octave are played at 32,767 beats per minute. This shows a way to get special sound effects in HyperTalk.

2. Enter

```
play "boing" tempo 16000 "c6w c c c c c c c c c c c c c c c"
```

in the Message box and press Return.

Result:

The laser buzz turns into a high-pitched horn. Use this trick to get the user's attention. If you like, you can install this in a button and put it in the special effects panel of the Sound Lab.

The *play stop* Command

Because HyperTalk sounds play in the background, once they start, they continue to play until all the notes are exhausted. You can stop the playing of the notes from the keyboard by pressing the Command and period (.) keys simultaneously. But what happens when you want to stop the sound from a script? Remember, your user may not be familiar with HyperCard's slam-on-the-brakes Command-. combination. The `play stop` command is provided to allow you to stop the sound from a script. Its syntax is `play stop`. Let's see an example of how to use `play stop`.

In chapter 13, "Advanced Math," you learned how to use the `cos` function to generate what sounded like random music. If you recall, the problem with that script was that when you tried to stop it, the music continued playing for some time, depending on how many notes had queued up. Let's see how we can improve that script with a `play stop` command so that we can halt it immediately with a single click of the mouse.

Example 10:

Go to your Sound Lab card, duplicate one of the special effects radio buttons, rename the button `Jazz`, and enter the following handler:

```
on mouseUp -- Jazz button
  put 0 into n
  repeat forever
```

continued

```
      put random(34) + three into fac
      repeat random(4) + 1 times
        play harpsichord tempo 300 55 + round(10 * cos(2*PI*n/fac))
        add 1 to n
        if the mouseClick then
          play stop              -- this stops the sound
          exit mouseUp
        end if
      end repeat
    end repeat
  end mouseUp
```

Result:

When you click the button, you hear what sounds very much like modern jazz. When you click again, the sound stops immediately.

The handler is similar to the previous one, except we have made it generate random numbers for the variable `fac`. This variable controls how many notes are in the repeating pattern. This repeating pattern is played between one and five times, depending on the result of the second random statement.

The script uses the statement `if the mouseClick then` to allow the user to click to stop. The `exit mouseUp` will end the handler when the next click occurs, regardless of where the mouse is. If that is all that is done, the script stops but the music continues. If you add the `play stop` command, the music stops when the script stops. You may want to go back and modify the sound scripts such as the Sweep button so that you can click to end the music instead of having to press the Command-. combination.

Sound Synchronization and *the sound*

So much for making music. What about using music with stacks? There are many applications in which you might want to have visual effects and cards presented so that they are synchronized with sounds. This is particularly true of custom sounds that you put into your stacks. We don't explain how you get digitized audio into your stack here, but we do see how to simulate a long audio sound and synchronize it with what is happening on the screen.

Consider a demo stack for a new product. You may want to have several cards, timed with the beat of a popular song, that present visual information about your product. You may want to have a voice speaking about the card that is being shown. This might seem easy at first, but it is not. The music's pace is fixed and can't be changed. If you want your card to dissolve just at the point when the voice is done or the music switches, you have to measure the length of time each sound takes and set up a `do nothing wait` loop in the card script that waits the approximate number of ticks. Very crude.

Another synchronization problem results because the Macintosh can't hold very many seconds of sound in memory at one time. This means you get lots of disk

activity if the sound is more than a few seconds long, because the sound manager is moving chunks of the voice in and out of memory. On a floppy, this disk activity causes the sound to break up and buzz. The only thing you can do is cut up your digitized sound into sections and play them back to back.

HyperTalk, however, provides a function called `the sound` that helps you synchronize sounds with cards. This function returns the name of the playing sound resource. When there is no resource being played, `the sound` returns the string `done`. Let's see how to use the sound to synchronize some cards with some sounds.

Example 11:

1. Go to your Practice card in the HyperTalk Bible stack, create a new button, name it `Sound.Sync`, and enter the following handlers:

```
on mouseUp -- Sound.Sync button
  global note
  put 20 into note
  repeat 10 times
    put (note + 1) mod 10 into note
    talk
    wait until the sound is "done" -- this does the magic
    visual effect dissolve to gray
    go to this card
  end repeat
end mouseUp


on talk
  global note
  play "boing" tempo 20 (50 - note) -- simulates a few secs of
music
end talk
```

   The on talk handler contains a `play` statement that is supposed to simulate a long amount of audio stored in a resource, such as someone talking (talk). This might also be a musical track, train whistle, explosion, whatever. Imagine that the slow boing is actually the Beatles singing "Eight Days a Week."

2. Click the button.

Result:

You hear a low boing sound effect, followed by the screen dissolving to gray. There is a second boing, lower in frequency and lasting longer. When it ends, the screen again dissolves to gray. This repeats until you press Command-. or ten boings are played, whichever comes first.

This example shows that the command `go to this card` is not executed until the boing sound is completed. Note, there is some silence at the end of the boing that causes a delay.

3. Open the button script and add two hyphens in front of the statement `wait until the sound is "done"` to disable it by commenting it. Click the button.

Result:

The card dissolves immediately into itself and repeats at maximum speed, while the sound occurs at its own rate. The two are out of sync. The 10 dissolves run out, but the sound continues.

Understand *wait until the sound is "done"*
The word "done" is not a HyperTalk constant like up or down, but is, rather, a string literal and therefore must be enclosed with quotation marks.

This last experiment shows that HyperCard is queuing up the sounds because it is proceeding ahead with processing the go to this card statements. By the way, if you change the repeat 10 times to repeat forever and run this script long enough, HyperTalk starts to slow down and the sound and dissolves actually come into sync.

Summary

This chapter covered all the advanced topics of sound generation. You learned the basics of standard musical notation, and looked at the language HyperCard uses to represent sound. You saw how to use the play statement to play simple music, and explored HyperCard's built-in voices. You learned about note duration, tone, and tempo. We showed you how to build a sound project script to make interesting special effects. You also looked at a simple technique to synchronize sound with visual effects.

Review Questions

1. Which of the following are legal forms of the play statement?

   a. play "a b c d e f g"
   b. play "farmerJohn" "60 70 80"
   c. play forbidden planet
   d. play "boing" tempo card field 5 "cq d e c c d e c e f gh"

2. Match the correct description with the corresponding play note statement syntax.

   a. a
   b. a#
   c. bbx.
   d. att
   e. c0
   f. fb
   g. c#6
   h. ebe

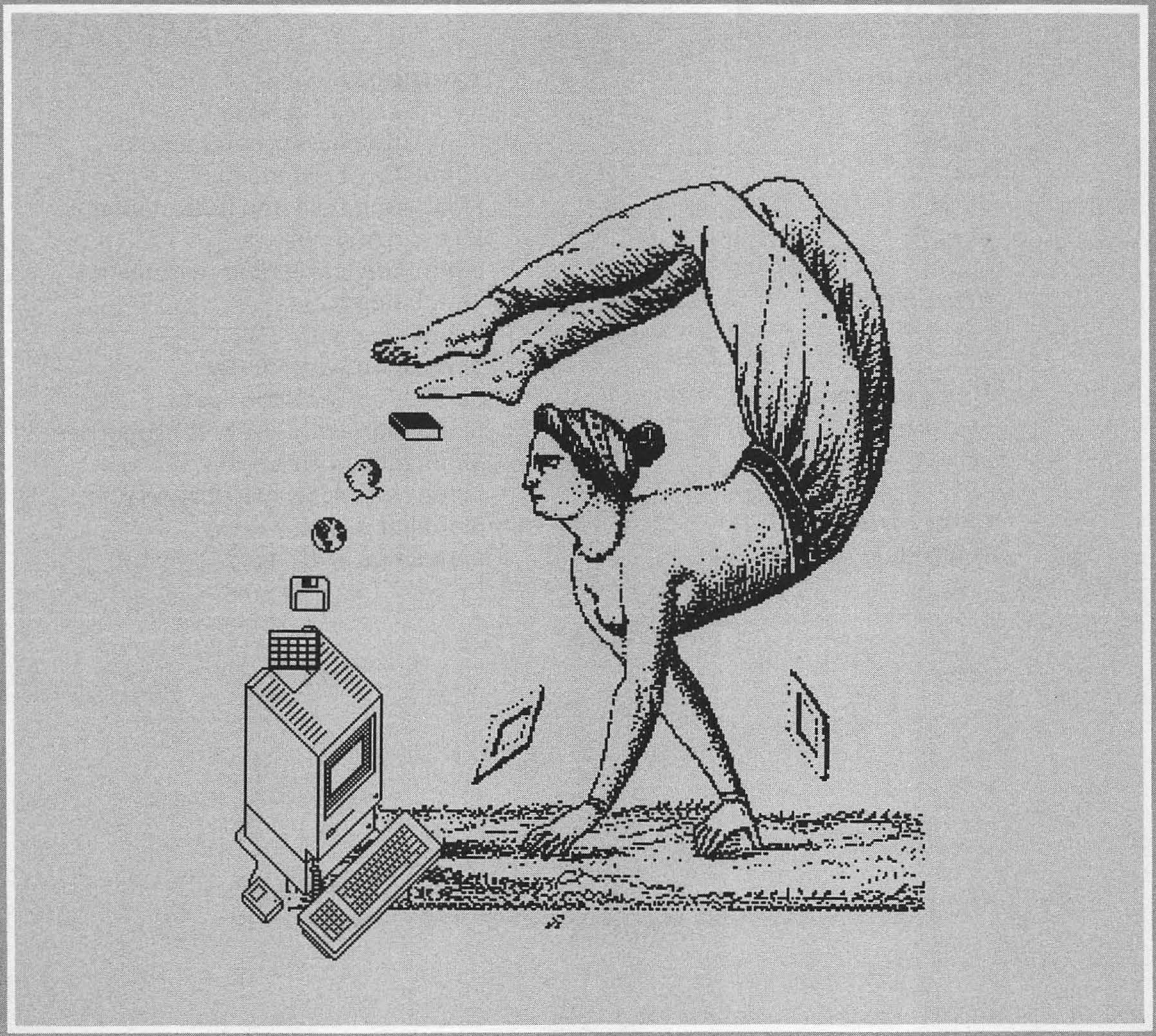
   i. C-sharp in the sixth octave
   j. lowest note
   k. A
   l. dotted B-flat 64th-note
   m. A-sharp triplet 32nd-note
   n. A-sharp
   o. E-flat eighth-note
   p. F-flat

3. Write play statements to do the following:

   a. play a 12-note upward rising scale starting at f3
   b. make a nice elevator bink sound
   c. make a horn buzz
   d. make a drain sound

Chapter 16

Simulation



Quick View

Commands
`type`
`choose`
`set`
`click`
`drag`

Functions
`the tool`

Properties
`rectangle`
`scroll`

Concepts
Controlling menus

Concepts *cont'd*
Typical use of `doMenu`
Simulating background menus
Simulating card menus
Simulating field and button menus
Rearranging objects
Simulating navigation with menus
Simulating painting
Simulating with `type`
Simulation with `choose`
Simulation with `the tool`
Simulation with `set` and properties
Simulation with `scroll`
Using the `rectangle` property
Simulation with `click`
Simulation with `drag`

Chapter 16

Simulation

You may have heard people say that anything you can do as a HyperCard painter or author, you can do as a HyperTalk scriptor in a script. Well, this is almost true. There are a few things scriptors can't do. We tell you about these limitations as they come up, but there are more things that scriptors can do that authors can't. In fact, simulation is the next best thing to AI (artificial intelligence).

In this chapter and the next, we show how to simulate the use of the painting and authoring tools in your scripts. This chapter concentrates on working with objects (stacks, backgrounds, cards, buttons, and fields). Chapter 17, "Graphics," demonstrates the painting tools available to you.

Because you can't use the command set without having some properties to "set," this chapter also covers the object (stack, background, card, button, field) properties. By the end of this chapter, you will have all the tools you need to create, manipulate, and delete HyperCard objects under script control.

The *doMenu* Command

You have already seen `doMenu` used in examples in this book. The command does exactly what its name implies. It simulates the user's pulling down a menu and selecting a menu item.

If you haven't used `doMenu` yet, try it now in the Message box. To do so, simply show the Message box, enter `doMenu "Card Info..."`, and press the Return key. The result shows the display of the same Card Info dialog box you would have seen had you pulled down the menu yourself.

Note: For the purposes of this chapter, we assume that every user's `userLevel` global property is set to 5 (Scripting).

In a script, you can use `doMenu` to simulate the selection of any menu item. You need to consider only two things before executing a `doMenu` command: the exact name of the item and whether this item is active right now. We have included the following list to answer both questions.

When `userLevel` is 1, active menu items are

Back, Find..., First, Help, Home, Next, Last, Message, Open Stack..., New Stack..., Page Setup..., Prev, Print Card, Print Report..., Print Stack..., Quit HyperCard, Recent, Save a Copy..., Undo

When `userLevel` is 2, active menu items are

The items above and Clear Text, Copy Text, Cut Text, Delete Card, New Card, Paste Text

When `userLevel` is 3 and a general tool is selected, active menu items are

The items above and Background, Compact Stack, Copy Card, Cut Card, Delete Card, Delete Stack..., New Card, Open Stack..., Protect Stack..., Text Style... (when insertion point is in a field)

When `userLevel` is 4 or 5 and a general tool is selected, active menu items are

The items above and Bkgnd Info... (when button or field is selected), Bring Closer (when button or field is selected), Button Info (when a button is selected), Card Info..., Clear Button, Clear Field, Copy Button, Copy Field, Cut Button, Cut Field, Field Info... (when a field is selected), New Background, New Button, New Field, Paste Button, Paste Field, Send Farther, Stack Info...

When `userLevel` is 3+ and a painting tool is selected, active menu items are

Back, Background, Brush Shape..., Clear Picture, Copy Picture, Copy Card, Cut Picture, Cut Card, Darken, Delete Card, Draw Centered, Draw Filled, Draw Multiple, Edit Pattern..., Export Paint..., FatBits, Fill, Find..., First, Flip Horizontal, Flip Vertical, Grid, Help, Home, Import Paint..., Invert, Keep, Last, Lighten, Line Size..., Message, Opaque, Paste Picture (if a picture is on the Clipboard), Paste Text (if text is on the Clipboard), Paste Button (if a button is on the Clipboard), Paste Field (if a field is on the Clipboard), Pickup, Polygon Sides..., Power Keys, Prev, Quit HyperCard, Recent, Revert, Rotate Left, Rotate Right, Select, Select All, Text Style... (when the text tool is selected), Trace Edges, Transparent, Undo

doMenu Does DAs

The `doMenu` simulation command also accesses Desk Accessories stored in the Apple Menu. For example, `doMenu "Calculator"` is a way to enable the Apple four-function calculator. If the calculator is not on the menu, you get an error message `Can't find menu item`. Unfortunately, this is a

fatal error, so you must find some way to guarantee that the menu item is on the Apple DA menu before executing `doMenu`. One way is to use an ask box to tell the user to check whether the DA exists.

Simulating Stack Menu Items

Some menu items in the File menu apply to an entire stack. You are already familiar with most of these. Open Stack lets you open another stack no matter where it is, New Stack makes a new stack that can have the same background as the current card, Save a Copy makes a copy of the current stack but doesn't open it, Compact Stack removes all the free space in a stack and leaves more space on the disk for new work, and Delete Stack is the electronic equivalent of throwing the entire stack of cards into the trash. These items work for the HyperCard author in the same way as they work for the HyperTalk scriptor.

Protect Stack displays a dialog box that lets you restrict the use of a stack in a number of ways. You might want to show the Protect Stack dialog box to users the first time they open a stack that is likely to contain sensitive data.

Print Stack also works on an entire stack. Print Report works on either a single background or an entire stack. These commands work the same in a script as they do when you select them with the mouse. There is one final stack-related item on another menu: Selecting Stack Info on the Object menu shows a dialog box with information on the current stack. It also allows you to edit the stack name and gives access to the Script Editor.

You can simulate stack manipulation menu selections to advantage in a HyperTalk script. If you are working on an important stack, you want to make sure you always have a backup copy just in case something mysterious happens to your working copy. One way to do this is to design a quitting routine that first compacts the stack, to get rid of any free space, and then makes a copy of the now compressed stack before it closes it. There are at least three reasonable ways to close a stack depending on what you want to do next. In the following example, we display an answer box giving the user the choice of opening another stack, going Home, or quitting HyperCard. As a final friendly touch in a script that requires user input between dull periods of computer activity, we include some audio feedback to alert the user that some input is required. Does this sound like a lot of work? Not in HyperTalk it's not.

Example 1:

1. Go to the last card of your HyperTalk Bible stack.
2. Add a new background to the stack and name the first card `Simulations`.
3. Add a new card button to the Simulations card and name it `Out-of-here`.
4. Enter the following handler in the Out-of-here button script:

```
on mouseUp -- Out-of-here button
  doMenu "Compact Stack""
  play "harpsichord"
```

continued

```
  doMenu "Save a Copy..."
  play "harpsichord"
  answer "Quit to..." with "Finder" or "Home" or "Open"
  if it is "Finder"
  then doMenu "Quit HyperCard"
  else if it is "Home"
  then doMenu "Home"
  else doMenu "Open Stack..."
end mouseUp
```

5. Click the Out-of-here button.

Result:

First, you see the spinning beach ball as HyperCard compresses the stack. Then, you hear a tone and are asked to approve the saving of the copy (you can click OK to approve or Cancel if you don't want a copy). Finally, you hear another tone and are asked how you want to quit the stack.

Simulation is simple and powerful. In this short example, you can either save a copy or not; you can quit to the desktop or to Home or to another stack, or you can cancel the quit and stay where you are. (You do have to compact the stack unless there isn't enough room on the disk.) How you abort a quit isn't as obvious as it could be because the `answer` command is limited to three options, but if you choose Open you get the Open Stack dialog box, which can be canceled, dropping you back to where you started, with the stack compacted and possibly copied.

Ideally, this handler would be in the Home stack's stack script where it could be accessed at any time. To do this, type a `user` command (see chapter 14, "Advanced Commands and Functions") into the Message box to run the handler located in the Home stack. The next example shows you how to do this.

Example 2:

1. Enter the following handler in your Home stack's stack script.

```
on outa
  doMenu "Compact Stack"
  play "harpsichord"
  doMenu "Save a Copy..."
  play "harpsichord"
  answer "Quit to..." with "Finder" or "Home" or "Open"
  if it is "Finder"
  then doMenu "Quit HyperCard"
  else if it is "Home"
  then doMenu "Home"
  else doMenu "Open Stack..."
end outa
```

2. Enter `outa` in the Message box and then press the Return key.

Result:

First, you see the spinning beach ball as HyperCard compresses the stack. Then, you hear a tone and are asked to approve the saving of the copy (you can click OK to approve or Cancel if you don't want a copy). Finally, you hear another tone and are asked how you want to quit the stack.

Simulating Background Menu Items

Moving down the object hierarchy one step from stacks brings you to backgrounds. There are only three menu items related to backgrounds: Background, Bkgnd Info, and New Background. You have probably used Background, in the Edit menu, to edit the background layer of a card. Background toggles so that you can go back and forth between layers by repeatedly selecting it. You will most often need to go to the background when painting, but there is one use of background that falls into the scope of simulations. Moving a button or field from the card layer to the background layer requires that you first cut the object, then go to the background, and finally, paste the object.

Bkgnd Info and New Background are both located in the Object menu. Bkgnd Info is the way to access information about the current background and the script for that background. Selecting New Background adds a new card to the stack that does not share the background of the card before it. In fact, the new card's background is completely blank with no graphics, no buttons or fields, and an empty script. You use New Background in your scripts the same way you would use it as an author—it is the only way to insert a new, empty background in a stack. The Bkgnd Info dialog box information is available to the scriptor in other (more easily manipulated) forms, but for those occasions when you want to show the user the dialog box, the option is there.

Simulating Card Menu Items

Print Card can always be selected from the File menu or by pressing Command-P. Most of the card items capable of simulation in a script are located in the Edit menu. New Card adds a card of the current background behind the current card. Delete Card removes the current card and shows the card behind it. Cut Card is the same as Delete Card except that it puts the removed card onto the Clipboard rather than into the trash. Copy Card puts the current card onto the Clipboard without removing it. You use these menu commands when you need to add new cards, remove old cards, or move an entire card from one place in the stack to another place in the same stack or in some other stack.

Now, comes the "shy" card item. The Paste Card menu item is only visible after a card has been cut or copied to the Clipboard; the Paste item varies to reflect the current contents of the Clipboard. This is the same way cut and copy work for pictures, buttons, fields, and text—as you will soon see. In case you are wondering why there are special case card items for copy and cut but not for paste, the answer is that cards are the only option that cannot be selected with the mouse. The Copy item changes to Copy Button when a button is selected and to Copy Text when text is selected, but you need a special menu item to copy cards. Paste, on the other hand, reflects the contents of the Clipboard so that the same item can be used for all the options including buttons, fields, text, pictures, and cards.

The final card item is Card Info on the Object menu. As with the other Info

items, Card Info displays a dialog box that provides information—some of which you can edit—on the current card and gives access to the Script Editor.

Of these items, probably the most useful to the scriptor are New Card, Delete Card, Paste Card, and Copy Card. There are three ways to add a new card to a stack: Use New Card to add a card with the same background as the current card but with empty background fields; use Copy Card and then Paste Card to add an exact copy of the current card, including the contents of the fields; or you can use New Background to add a card with a new background. In the following example, we do all three.

Example 3:

1. Go to the Simulation card of your HyperTalk Bible stack.
2. Add one background field and one card field to the Simulation card and type some text into each field.
3. Add a new card button to the Simulation card and name it `Card Adder`.
4. Enter the following handler in the Card Adder button script:

```
on mouseUp -- Card Adder button
     push card
     doMenu "Copy Card"
     doMenu "New Card"
     doMenu "New Background"
     doMenu "Paste Card"
     push card
     doMenu "Home"
     go next -- don't skip this as the first card in the Home stack
     -- is protected and will be hard to delete
  doMenu "Copy Card"
  pop card
  doMenu "Paste Card"
  pop card
end mouseUp
```

4. Click the Card Adder button.

Result:

The following cards are added to the stack: a card that shares the same background at the original card (note that it shares the background field but the field is empty), a card with a brand new, and empty, background, a card that is an exact copy of the original card, including objects in the card layer and the contents of the fields, and a copy of the second card of the Home stack.

We said there were only three ways to add cards to a stack, but the last method used by this script is a bit of a hybrid. Copying a card from another stack makes an exact copy of that card, but it also brings with it a background new to the current stack.

Let's use the other useful menu item, Delete Card, to get rid of all these cards. The technique is so simple that it doesn't require a full-scale example. What you do is write a script that goes to the next card, repeats a `doMenu "Delete Card"` command four times, and then returns to the current card.

Simulating Field and Button Menu Items

Because button items and field items are very similar, we describe the items that are common to both buttons and fields first and then discuss the items that are unique to fields or to buttons. The first batch of menu items to simulate can be found under the Edit menu. Cut, Copy, Paste, and Clear do double duty depending on which type of object is selected or, in the case of Paste, on which object is on the Clipboard. When using `doMenu` with these items, you need to keep track of what is selected and what is in the Clipboard, or you can accidentally try to do something that isn't possible and get an error message. For example, you can't copy a button when a field is selected or paste a field when a button is on the Clipboard. To see exactly what we mean, select a button, choose the Copy Button menu item to copy it onto the Clipboard, and then select a field. Now, pull down the Edit menu and you find the following items: Cut Field, Copy Field, Paste Button, and Clear Field. Enter `doMenu "Paste Field"` or `doMenu "Cut Button"` in the Message box, and you see that HyperCard can't understand what you want. To avoid confusing users with these error messages, keep track of what is selected and what is on the Clipboard and don't ask for menu items that aren't currently available.

Also in the Edit menu, you find the Text Style item. This item doesn't apply to buttons but is active when a field is selected or active. In Versions 1.0.1 through 1.2.2, the font, text size, text style (and the other text characteristics you can set in the dialog displayed by this item) apply to all text in a field and not just to a selected portion. Just because the Text Style dialog box is not available for buttons doesn't mean you can't change the text in a button. When we cover set and button properties, we show a way to do exactly that—something a HyperCard author can't do. For example, you can use a `doMenu "Text Style"` command to give the user an opportunity to choose text attributes the first time a field is used, or you can make a button that lets you set the text attributes of any field you click with the option key pressed.

Example 4:

1. Go to the Simulation card of your HyperTalk Bible stack.
2. Add a new card button to the Simulation card and name it `Field Style Edit`.
3. Enter the following handler in the Field Style Edit button script:

```
on mouseUp -- Field Style Edit button
  put " To edit text attributes click over a field " into msg
  wait until the mouse is down
  choose field tool
  click at the mouseLoc
  doMenu "Text Style..."
  choose browse tool
end mouseUp
```

4. Click the Field Style Edit button and then click a field. Change any of the text properties you wish and then click the OK button.

Result:

When the Text Styles dialog box disappears, you see the changes you asked for in the field you clicked.

On the Objects menu, you find Button Info and Field Info; New Button, and New Field. Button Info displays the dialog box you are probably familiar with and provides access to the Script Editor for the chosen object. New Button makes a new round-rectangle-style card button named New Button. New Field makes an unnamed transparent field. You will use these menu commands in your scripts to make new buttons and fields with default properties; and to show the user the respective Info dialog boxes.

There are two more items on the Objects menu that are shared by both object types. Bring Closer and Send Farther apply to the selected object and are unavailable if nothing is selected. What these items do is best understood if you think of each button and field on a card as existing on its own transparent layer. Bring Closer moves the selected object and its layer up, layer by layer, past other buttons and fields until it reaches the top, but not necessarily the absolute top. Background objects can never rise higher than the card graphic layer that is below the bottom card object.

Send Farther works the same way with the same limitations. Objects can be moved back, but card layer objects can never fall below the card graphic layer. The object numbers tell you the vertical order within each type of object. Button 1 is the bottom button and button 2 is above it. If you select button 2 and choose the Send Farther menu item, the number of the button changes to 1 . . . or will it? The most confusing thing about these two menu items is that on a card with many buttons and fields, you can use Send Farther several times without its seeming to have any effect. Suppose that you want to send button 2 behind button 1. If there are only buttons on the card, you only need one Send Farther, but if there happen to be a couple fields in layers between the two buttons you need several Send Farthers to achieve the same end. In this case (as shown in Figure 16-1A) button 2 was actually object 4. The first Send Farther made button 2 into object 3 (Figure 16-1B); the second Send Farther made it object 2 (Figure 16-1C); only the third Send Farther made it object 1 and button 1 (Figure 16-1D).

For those occasions when you have objects you need to move up or down on a crowded card, a script can be used to automate what can be a tedious manual process.

Example 5:

1. Open the Simulation card of your HyperTalk Bible stack.
2. Add 2 new card fields and then a card button to your HyperTalk Bible stack. Name the card button `Sender`.
3. Enter the following handler in the Sender button:

```
on mouseUp -- Sender button
  doMenu "New Button"
  -- the technique below will work even if there are more than
  -- one New Buttons as long as it's the top one you want to
  -- send.
  repeat until the number of button "New Button" is 1
    doMenu "Send Farther"
    put "" after the message box -- Option-8 on the keyboard
  end repeat
```

continued

```
  put "The New Button" is now card button 1" after the msg
end mouseUp
```

4. Click the Sender button.

Result:

A new button appears on the screen and is moved back, layer by layer, until it is behind all the other card buttons. A bullet appears in the Message box for each Send Farther command. Count the bullets and compare to the number of card buttons (should be 5 buttons and 6 bullets).

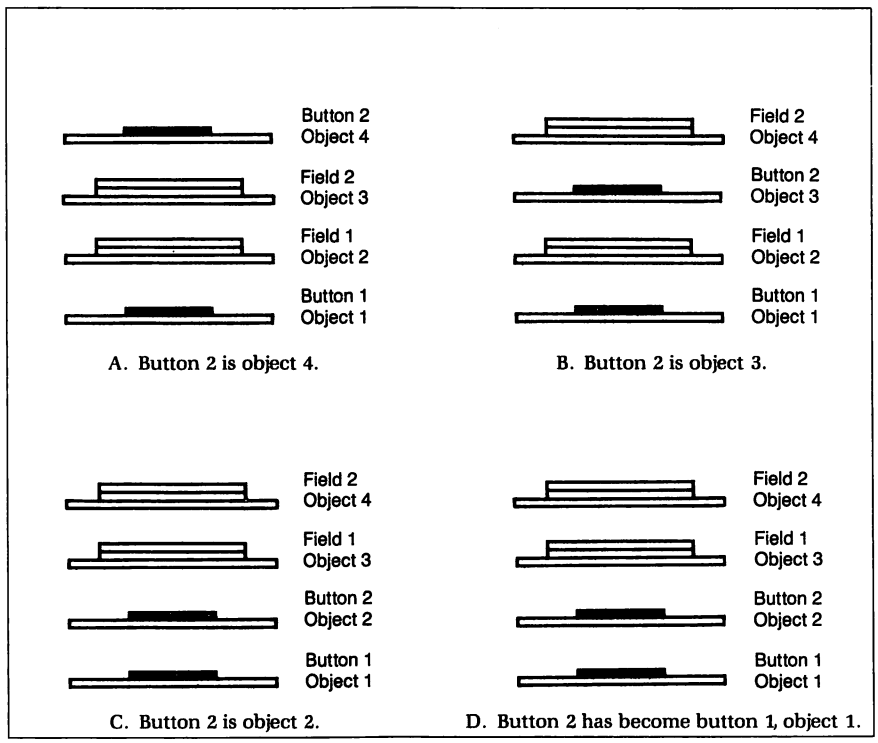


Figure 16-1. Rearranging object layers.

Send and Bring All the Way

Starting with version 1.2, you can send an object all the way to the back or bring it all the way to the front with a single command. At the keyboard, you do this with Command-hyphen (-) or Command-+, and in a script, you do it with `type "-" with commandKey,shiftKey` or `type "+" with commandKey,shiftKey`. Here is a short script that brings a specified object to the front no matter how many objects are above it.

```
on mouseUp -- button script
  select button "WayBack"
  type "+" with commandKey,shiftKey
```

continued

```
    set the name of button "WayBack" to "UpFront"
end mouseUp
```

Simulating Navigation Menu Items

Because all the Go menu items are also `go` command options (`go first`, `go last`, etc.) there is no reason to simulate the use of this menu with `doMenu`. See chapter 2, "Objects," for information on the `go` command.

Simulating Painting Menu Items

The painting tools, the Pattern Palette, Import Paint, Export Paint, Undo, and the items in the Paint and Options menus are covered in chapter 17, "Graphics."

Simulating Miscellaneous Menu Items

Other menu items worth simulating are found in the File menu. Quit HyperCard, which we have already simulated in Example 1, is useful for returning to the desktop.

Moving over to the Edit menu, Undo undoes object repositions, resizes, and clones, but it can't, at least in versions 1.0.1 through 1.2.2, bring back an object you deleted.

In the Go menu, the `message` command toggles the Message box; Find either shows the Message box with Find "—" entered so that you can quickly type the word you want to find; or, if the Message box is already shown, it simply puts the Find "—" prompt into it. Doing Find a second time does not hide the Message box, but a `doMenu "Message"` does. All Find does is put the HyperTalk `find` command into the Message box. You might use a `doMenu "Find"` command in a script to let the user search for some text string, but there are HyperTalk `find` command options that offer more control.

The desk accessories under the Apple are also accessible with the `doMenu` command. The problem, of course, is knowing what is there because this menu is customizable by each Macintosh user. If you are doing scripts for yourself, you may have a good idea what is available, and you can call your desk accessories from your scripts. If you are writing for general users in the Mac community, however, you have to be more cautious. You can assume that most, but not all, people have Control Panel, Scrapbook, Chooser, and Key Caps. If you are using the Scrapbook to move things into or out of HyperCard, it might be a good idea to have a little button to automate opening the Scrapbook because this item is at the end of the alphabetized list. This requires only a one line script after all: `doMenu "Scrapbook"`. Even better, you can put this into a user command handler in your Home stack's stack script and access it through the Message box by typing `scrap` and pressing the Return key. The entire handler is

```
on scrap
  doMenu "Scrapbook"
end scrap
```

If you are going to use sound, you might consider adding a `doMenu "Control Panel"` command to encourage the user to raise the volume setting.

Before we move on, we should point out that many menu items do something beyond the obvious actions we have discussed above. Many of them also send messages of various kinds that you may want to trap or stifle. One example is the Quit HyperCard item which results in the following messages being sent: `closeCard`, `closeBackground`, `closeStack`, and `quit`.

Simulation with the *type* Command

In chapter 11, "Strings," you learned about the `type <string>` command, which is the equivalent of the user typing the same string at the keyboard. Recall that this gave a nice typewriter effect—as if some patient person was typing to the screen. It also allows you to show the user how you would enter information interactively. The `type` command is another kind of simulation that can be used in several ways. If you want to put text into a field in a more dramatic way than with the `put` command (which does it blindingly fast), you can use `type` to simulate slowly typing into the field. The only tricky aspect of doing this is getting the insertion point cursor into the right field. You can use `click at` for single line fields but only `type tab` works correctly for all fields. You can figure how many tabs to type by discovering the number of fields on the card.

Example 6:

1. Open the Simulation card of your HyperTalk Bible stack.
2. Add 2 new card fields and then a card button to your HyperTalk Bible stack. Name the card button `Type Test`.
3. Enter the following handler in the Type Test button:

```
on mouseUp -- Type Test button
  put the number of fields into bkgndCount
  put the number of card fields into crdCount
  put "This is,a test,of typing,to fields" into textVar
  repeat with i = 1 to bkgndCount + crdCount
    type tab
    type item i of textVar
  end repeat
end mouseUp
```

4. Click the Type Test button.

Result:

The handler determines the number of fields, both card fields and background fields (should be 4 for this card), and then tabs to each field and types some text.

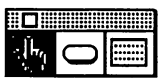
Another way to use `type` is as an alternative to `doMenu` for menu items with command key alternatives. For example, `type "o" with commandKey` is the equivalent of `doMenu "Open"`. Likewise, `type "c" with commandKey` can replace `doMenu "Copy Button"`, `doMenu "Copy Field"`, `doMenu "Copy Picture"`, and `doMenu "Copy Text"`. For the most part, whether you decide to use `doMenu` or `type` is a matter of style and personal preference, but there are some cases where `type` can be a real advantage. If you need to copy and paste both buttons and fields, you have to write two copy routines and two paste routines if you use `doMenu`. If you used `type`, however, you only need one of each. On the other hand, the `doMenu` routines are much easier to read than the cryptic `type` routines.

Simulation with the *choose* Command

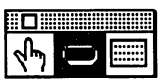
The `choose` command is similar to the `doMenu` command but is used just to select specific items from the Tool Palette. The `choose` command was originally introduced in chapter 3, "Audio and Visual Effects."

The Tool Palette itself is divided into two areas: The top row of three tools are the general tools, which we cover in this chapter, and the remaining 15 tools are the painting tools, which are covered in chapter 17, "Graphics."

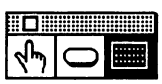
The general tools are the browse tool, the button tool, and the field tool.



The browse tool is the standard HyperCard tool for pushing buttons and entering text fields. No matter what tool is currently chosen, you can activate the browse tool at any time with the command `choose browse tool`.



The next tool in the general tool palette is the button tool you use to select, reposition, clone, delete, and resize buttons. As you probably already know, you can also use the button tool to open the Button Info dialog box (by double clicking a button) and to open a button's script by double clicking the button with the Shift key pressed down.



The final general tool is the field tool, which does for fields everything the button tool does. Before you look at an example using `choose`, we introduce a function that is the `choose` command's best friend.

These tools can be used (in conjunction with the `drag` command) in your scripts to simulate doing all the things you normally do with the mouse.

Simulation with *the tool* Function

It's often a good idea to find out how things stand before you start making changes to them. With the function, `the tool`, you can discover what tool is currently in use

before you go to the trouble of choosing a new one. The syntax for the tool is shown in Figure 16-2.

```
the tool
tool()
```

Figure 16-2. The `the tool` function syntax.

You can also save the tool that was active when a script began and restore it again at the end. For example, if the browse tool is active and your handler alters buttons by selecting the button tool, it can leave the button tool selected and the user confused. One way of avoiding this problem is to put the value returned by `the tool` into a variable like `toolVariable` at the beginning of your handler and then add a `choose toolVariable` statement at the end of the handler. This leaves things the way you found them.

Example 7:

1. Go to the Simulation card of your HyperTalk Bible stack.
2. Name the New Button from the last example `Tool Saver`.
3. Enter the following handler in the Tool Saver button script:

```
on mouseUp -- Tool Saver button
  put "Starting tool is" && the tool && "" into msg
  put the tool into toolKit
  choose button tool
  put "then" && the tool && ""after msg
  if the textFont of me is "Chicago" then
    set the textFont of me to "Geneva"
  else
    set the textFont of me to "Chicago"
  end if
  choose toolKit
  put "and finally" && the tool after msg
end mouseUp
```

4. Click the Tool Saver button.

Result:

`Starting tool is browse tool` appears in the Message box followed rapidly by `then button tool`, the button's font is changed, and then `and finally browse tool` appears in the Message box.

The second major simulation use for the `tool` function is logical expressions (those that are either true or false in `if then` control structures). We demonstrate this use in the next example.

Here is an example that lets you cycle through the three general tools (browse, button, field) by repeatedly hitting the Enter key. This is much easier than having to

constantly go to the Tool Palette to change tools. It would be hard to overestimate the amount of time this little handler saves you if it is placed into your Home Stack's stack script where it always is available. For the purposes of this exercise, we only put the script into the HyperTalk Bible stack's stack script so that you can try it.

Example 8:

1. Go to the Simulation card of your HyperTalk Bible stack.
2. Open the stack script and enter the following handler:

```
on enterKey
  if the tool is "browse tool"
  then choose button tool
  else if the tool is "button tool"
  then choose field tool
  else choose browse tool
  pass enterKey
end enterKey
```

3. Tear off the Tool Palette by dragging it from the Tool menu or by typing Option-Tab.
4. Press the Enter key. Pause. Repeat.

Result:

Each time you press the Enter key, a different tool is selected. Repeatedly pressing Enter cycles through the three general tools.

This handler uses an unconventional control structure (which was discussed in chapter 9, "Ifs") to handle three options in an elegant fashion. This control structure is so flexible that you can even extend the example handler so that it cycles through the paint tools as well as the general tools. Cycling through so many tools probably is not practical, but you can customize the script to select the tools you use most often.

Simulation with the *set* Command and Object Properties

The `set` command is used to change the property settings of many object properties. To find out what a particular property is, use either the `get` or the `put` command, as in `get the name of button 1` or `put id of card field 1 into msg`.

Properties are just like functions and can be used in all the ways that functions are used. (See Table 16-1.) In the case of object properties, using `set` and either `get` or `put` simulates what the HyperCard user does when opening Info dialog boxes and making changes. For example, if you followed the instructions in Example 1, when you select the Card Info menu item, you can see that the name of the card is Simulations, and you can change it to SimCard by typing that string into the name field. In a script, you could do the same thing with `get name of this card` and `set name of this card to SimCard`. Besides the card property `name`, there are also `number`, `the id`, and `script` properties. You can't set the `number` or `the id`, nor can you change these

properties from the dialog box. The `script` property can be used to either copy a card's script into a container, as in `get script of card 2`, or to replace a card's script with the contents of a container, as in `set script of card 1 to it`.

Table 16-1. Stack, Background, and Card Properties

Stack Properties	Background Properties	Card Properties
name	name	name
script	the number	the number
the size	the id	the id
the freesize	script	script

If you want to show the Script Editor, as you would do manually by clicking the Script button in the Card Info dialog box, you must use the `edit script` command. The syntax for `edit script` is shown in Figure 16-3.

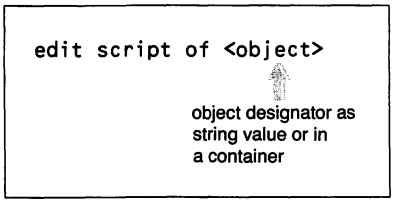


Figure 16-3. The `edit script` command syntax.

While you are working on a card's script, you could open the Script Editor quickly by using a Card Script card button with the following short script in it.

```
on mouseUp
  edit script of card this card
end mouseUp
```

Example 9:

1. Go to the Simulation card of your HyperTalk Bible stack.
2. Add a new card button to the Simulation card and name it `Card Button`.
3. Enter the following handler in the Card Button button script:

```
on mouseUp -- Card Button button
  get the short name of this card
  put "The name of this card is" && it into message box
  get the short id of this card
  put "; the id # is" && it after message box
  set the name of this card to "SimCard"
  wait 3 seconds
```

continued

```
    edit script of this card
  end mouseUp
```

4. Click the Card Button button.

Result:

The name and id number of the card is displayed in the Message box and then the Script Editor opens. Note that the card name at the top of the editor dialog box confirms that the card's name has been changed.

Stack and Background Properties

The stack and background properties are similar to the card properties except for `size` and `freeSize`. The `size` property reports the size on disk of the specified stack in bytes. The `freeSize` property tells the number of bytes of unused stack disk space that would be freed if you compacted the stack. This is a very handy fact to know, and you can use it to make a little closeStack handler that automatically compacts the stack if there is more than a set amount of free space in the stack. The next example shows you one way to do this.

Example 10:

1. Open your HyperTalk Bible stack and enter the following handler in the stack script:

```
on closeStack
  if the freesize of this stack > 10000 then
    put "Stand by: Compacting" into the msg
    doMenu "Compact Stack"
  end if
end closeStack
```

2. Look in the Stack Info dialog box and see whether there is more than 10KB of free space in the stack. If there is, go to the Home stack. If there isn't, you have to wait until you've built up some free space to see this work.

Result:

When you leave the stack, if there is more than 10KB of free space, you see the `Stand by: Compacting` message in the Message box and then the beach ball. If there is less than 10KB of free space, nothing happens.

Button and Field Properties

The button and field properties offer many more options than the card, background, and stack properties do. (See Table 16-2.) The most important button properties for simulation are the `location (loc)` and the `rectangle (rect)`. To refresh your memory, the `loc` of a button or field is the coordinate point at the center of the object. The command `get loc of button 1` returns a pair of numbers like 200,125 (horizontal,vertical) in the variable `it`. The `rect` of a button or field is the rectangle

that forms the outer boundary of that object. The command `get the rect of button 1` returns four numbers like 100,100,300,150 (left,top,right,bottom, commas included)) in the variable `it`. The importance of `loc` and the `rect` becomes more evident later in the chapter when we start using the commands click `at` and `drag`.

Table 16-2. Button and Field Properties

Button Properties	Field Properties
autoHilite	the id
hilite	loc
icon	name
the id	the number
loc	the rect
name	script
the number	scroll
the rect	showLines
script	style
showName	textAlign
style	textFont
textAlign	textHeight
textFont	textSize
textHeight	textStyle
textSize	visible
textStyle	wideMargins
visible	

If you select the Button Info menu item in the Object menu, you notice in the dialog box that you can see the name, type of button, number, ID number, style, and whether Show name and Auto hilite has been selected. From this dialog box, you also have the option of changing the button's icon, linking the button to some other card, and opening the Script Editor.

How much of this can you simulate in a script? You already know about `name`, `number`, and `id`. The `name` property returns the words `card button` or `bkgnd button` followed by the actual name of the button, for instance, `card button "New Button"`. Even if the object is unnamed, the first two words tell you the type, although what follows is an ID number rather than a name, `bkgnd button id 3`, for example. The command `get style of button <designator>` tells you the style of the button specified, and you can change that style with the command `set style`. You can `get` and `set` the `showName` and `autoHilite` properties. You can change the icon with `set icon`, and you can open the script with the command `edit script`.

As a HyperTalk scriptor, you can do even more. The command `set visible of button 3 to false` makes that button disappear. The button is still there; it is just hidden, much as the Message box can disappear without losing its contents. To show the button again, `set visible of button 3 to true`. Alternatively, you could use the commands `hide button 3` and `show button 3`, which have the effect of setting the

`visible` property to false and to true, respectively. Suppose that you have a button which you think would look much better if only it were inverted with white text or a white icon on a black background. All you need to do is `set the hilite of button 1 to true`, and it is inverted until the next `autoHilite` or until `hilite` is set to false. To keep the button inverted, even after an `autoHilite`, be sure to put the command to `hilite` the button in the button's script so it is called every time the button is used.

Example 11:

1. Open the Simulations card of your HyperTalk Bible stack.
2. Enter the following command in the Message box:

```
set the autoHilite of button "Type Text" to true
```

3. Click the Type Text button.

Result:

The first click inverts the button for an instant before it returns to normal.

4. Edit the mouseUp handler in the Type Test button as follows:

```
on mouseUp -- Type Test button

 set the hilite of me to true -- new line

  put the number of fields into bkgndCount
  put the number of card fields into crdCount
  put "This is,a test,of typing,to fields" into textVar
  repeat with i = 1 to bkgndCount + crdCount
    type tab
    type item i of textVar
  end repeat
end mouseUp
```

5. Click the Type Test button. Repeat.

Result:

The first click inverts the button. The second click briefly inverts the inverted button (equals normal) before it reverts to highlighted.

As you may have noticed in Example 7, you can change a button's text attributes. Tired of plain Chicago 12? Type the following commands into the Message box one at a time and you end up with outlined Venice, 14 point button text.

Example 12:

1. Open the Simulation card of your HyperTalk Bible stack.
2. Find button 1 (it should be the Tool Saver button) and make sure its name is shown.
3. Enter the following line in the Message box:

```
set textFont of button 1 to Venice
```

Note: if you don't have the Venice font in your system, substitute some other font.

4. Enter the following line in the Message box:

```
textStyle of button 1 to outline
```

5. Enter the following line in the Message box:

```
set textSize of button 1 to 14
```

Result:

The font of button 1 changes to Venice, then the style changes to outline, and finally, the size changes to 14.

Opening a button's script is pretty easy—just choose the button tool and double click the button with the Shift key pressed down. There is an even easier way to do this with a little scripting. Because you can use the edit script command to show the editor, you can add a couple lines of code to the top of a button you are working on to make it show its own script. Here is the code to add to the top of the mouseUp handler:

```
if the optionKey is down then
  edit script of the name of me
  exit mouseUp
end if
```

Then, if you clicked the mouse button with the option key down, the script would be shown. The fragment above runs, and the rest of the handler won't run unless the option key is down. If you want the rest of the handler to run immediately after you've finished looking at or editing the script, reduce the code to the one line `if the optionKey is down then edit script of the name of me`.

The field properties replace all the Field Info dialog box options: `name`, `number`, `id`, `style`, `wideMargins`, `showLines`, `lockText`, and the font properties `textAlign`, `textFont`, `textHeight`, and `textStyle`.

The scroll *Property*

The `scroll` property controls the display of a field's text in the field's display area. A scrolling field works by moving the displayed text up and down in pixel increments. The top pixel in a field is the 0 scroll, and a scrolling field normally opens with its scroll set to 0. You can test this with the command `get scroll of <field>`. If the text height of a field is 16, to move the text up one line, the field has to scroll 16 pixels. If you click the down arrow once and then get the scroll again you find that the scroll, which really means the top pixel displayed in the field, is now 16.

Have you ever looked at text scrolling automatically in other programs, often in the credits or in the help screen, and wished that you could do such esoteric things? Now you can. Make a scrolling field that stretches across the screen and hides the scroll bar with an opaque button.

Example 13:

1. Go to the Simulation card of your HyperTalk Bible stack.
2. Add a new button and name it `Fancy Scroll`.

3. Enter the following handler in the Fancy Scroll button script:

```
on mouseUp -- Fancy Scroll button
  doMenu "New Field"
  choose button tool
  drag from 494,101 to 511,198 with commandKey
  set the style of button the number of buttons to opaque
  choose field tool
  set the rect of card field the number of card fields to ¬
  0,100,512,200
  set style of card field the number of card fields to scrolling
  -- the next line gets a copy of this script and puts it into the
  -- variable it
  get script of me
  -- the next line transfers the script from it to the new field
  -- container (not the script)
  put it into card field the number of card fields
  choose browse tool -- get rid of field tool lines
  repeat with n = 1 to 51
    -- the next line moves the text in 5 pixel increments
    -- if you are using a 68020 machine you should substitute 1 for 5
    set scroll of card field the number of card fields to n * 5
  end repeat
  put "Fini" into message box
end mouseUp
```

4. Click the Fancy Scroll button.

Result:

You see the scrolling field (and the button that conceals the scroll bar) created, transformed, and then filled with text. After all that is done, HyperCard slips into Browse mode and then starts scrolling until all the text disappears off the top of the field and the word `Fini` appears in the Message box.

How would you use something like this? Perhaps when you are providing user information, or for stack credits. With a little more effort, you could offer different scroll rates and, of course, a down scroll control to complement the up scroll. To improve the appearance of this example, consider setting `lockScreen` to true before creating the field and button and back to false after the browse tool is chosen.

The visible *Properties*

We can't leave this topic without covering one of the most useful field properties: `visible`. The `set visible of field <designator>` command works the same way the `button` command works but is even more useful for fields. One of the great features of HyperCard is the pop-up field. You can make a field of any style and size, and hide it on the card so that it is shown only when a user requests it—by clicking a button perhaps.

Example 14:

1. Go to the Simulation card of your HyperTalk Bible stack.

2. Make a new card field, name it `PopUpField`, and select the Shadowed style option.
3. Make a new card button and name it `Hide Field`.
4. Enter the following handler in the Hide Field button:

```
on mouseUp -- Hide Field/Show Field button
  if visible of card field "PopUpField" is false then
    set visible of card field "PopUpField" to true
    set name of card button "Show Field" to "Hide Field"
  else
    set visible of card field "PopUpField" to false
    set name of card button "Hide Field" to "Show Field"
  end if
end mouseUp
```

5. Enter some random text in the field.
6. Click the Hide Field button.

Result:

The PopUpField disappears and the name of the button changes to Show Field.

7. Click the Show Field button.

Result:

The PopUpField reappears and the name of the button changes to Hide Field.

In this handler, the `line set visible of card field "PopUpField" to true` could be replaced by the functionally identical `show card field "PopUpField"`, and the line `set visible of card field "PopUpField" to false` can be replaced by the functionally identical `hide card field "PopUpField"`, but the property `visible` is still required to supply the current status of the field. Another possibility is the command `set visible of card field "PopUpField" to not visible of card field "PopUpField"`. The beauty of this last approach is that the same line can be used for both showing and hiding the field as it sets `visible` to the opposite of whatever it is. This allows you to substitute the following handler for the one in Example 14.

```
on mouseUp -- Hide Field/Show Field button
  if visible of card field "PopUpField" is false
  then set name of card button "Show Field" to "Hide Field"
  else set name of card button "Hide Field" to "Show Field"
  set visible of card field "PopUpField" to not visible of ¬
  card field "PopUpField"
end mouseUp
```

Version 1.2

Using the New Rectangle Properties and *is within*

Prior to Version 1.2, you could only get the `rect` of an object that returned four comma-separated numbers (Left,Top,Right,Bottom). To work with the individual numbers, you had to use chunk expressions such as

```
put item 4 of the rect of field 1 - item 2 of the rect ¬
of field 1 into msg
```

This statement puts the height into the Message box. Now, you can refer to each element directly with the new properties: `left`, `top`, `right`, `bottom` (`bot`). You can also refer to the two coordinate pairs that indicate the top-left and bottom-right corners of a rectangle with the properties: `topLeft` and `botRight`. With these properties, you can replace the previous statement with

```
put bottom of field 1 - top of field 1 into msg
```

That's not all. There are two more properties: `height` and `width`. With the `height` property, you can replace the previous statements with `put the height of field 1 into msg`. The `width` property returns the measure of the right side minus the left side.

These new properties can be set as well as read, which means that you can not only get information about the placement and size of your buttons and fields more easily than before, but you can also rearrange them on the screen more easily. Placing a button or field on the screen by setting the `loc` requires you to calculate what the `loc` has to be to get the border of the object where you want it. Previously, for example, if you wanted the top-left corner of button 1 to be at 100,100, you had to use the following command:

```
set the loc of button 1 to 100 + ¬
((item 3 of button 1 - item 1 of button 1)/2),100 + ¬
((item 4 of button 1 - item 2 of button 1)/2)
```

Now, you can replace that complicated statement with the command `set the topLeft of button 1 to 100,100`. That's progress! Setting the `height` leaves the center of the object (`the loc`) unchanged, but alters both the top and the bottom. Similarly, changing the `width` alters the `right` and `left` but leaves the center as before.

That's still not all. There is a new logical operator called `is within` that returns true if a specified point is within a specified rectangle. Previously, you had to determine whether the horizontal coordinate was to the right of the left side and to the left of the right side and then make sure that the vertical coordinate was above the bottom side and below the top side. Now, all you need is something like this:

```
if the clickLoc is within "20,100,30,110" then beep
```

One application for this operator is a situation in which you want to change the appearance of the cursor when it is over a portion of the screen. This requires that you keep program flow inside a handler where you can control things until the mouse leaves the specified area. You can define the screen area with a transparent button and use a `mouseWithin` message to change the cursor, but then you need either a `wait` command or a `repeat` loop to prevent the cursor from reverting to the hand cursor (when the handler ends). Here is what the handler would look like:

```
on mouseWithin
  set the cursor to plus
  wait until the mouse is down or not (the mouseLoc is within the rect¬
  of target)
  if the mouse is down then play harpsichord
end mouseWithin
```

This handler would give you a plus cursor when the mouse was inside the button, play a sound when the mouse button was clicked, and end when the mouse cursor left the button. Note that the parentheses are mandatory with `not... is within` and prevent HyperCard from trying (and failing) to change the truth value of the coordinate pair returned by `the mouseLoc` (which makes no sense). These additions to the language are significant improvements and save you time and trouble whenever you have to manipulate buttons or fields.

Another version 1.2 improvement is the ability to show button and field outlines by pressing Command-Shift-Option and then opening a field or button script by clicking it with those keys pressed.

Simulation with the *click* Command

Now that you can use `choose` to select tools and simulate selecting menu items and typing, the next step is to add the basic mouse capabilities: clicking and dragging. The `click` command was introduced in chapter 7, "Mouse", so you should already know that it allows you to simulate a mouse click at the coordinate location specified and, optionally, simulate clicking with the option key, command key, or Shift key modifier keys pressed down. You can even combine the three modifier keys in the same command statement.

The `click` command, along with the `loc` property, gives you the ability to select objects from scripts as if the user had clicked them (provided you know the object's location). Once an object is selected, you can use the `doMenu` command to show information about it, make copies of it, delete it, and so on. In previous examples, we always worked with a new object when copying and pasting because new objects are automatically selected when you first make them. In the next example, we use some of our new resources to make copies of existing objects.

Example 15:

1. Go to the Simulation card of your HyperTalk Bible stack.
2. Make a new card button and name it `Click Copy`.
3. Enter the following handler in the Click Copy button:

```
on mouseUp -- Click Copy button
  put the tool into toolVar
  choose button tool
  click at the loc of card button "Click Copy"
  doMenu "Cut Button"
  doMenu "Background"
```

continued

```
    doMenu "Paste Button"
    doMenu "Button Info..."
    choose toolVar
end mouseUp
```

4. Click the Click Copy button.

Result:

The Click Copy button is selected, cut, and copied to the background layer. At the end of the handler, the Button Info dialog box for Click Copy is shown, confirming its new background button status.

This may seem a simple-minded job until you consider how tedious it would be to manually move a number of objects from one layer to the other. More than one person (sigh) has started their HyperTalk career with a really ambitious database, with lots of objects on a card, only to realize halfway through that everything was in the wrong layer. In the Summary Scripts section at the end of this chapter, you can find an example of the previous technique extended to multiple objects.

Version 1.2

Using *select* with Fields and Buttons

Starting with Version 1.2, you can select a button or field directly without using the `click` command. Instead of writing

```
choose button tool
click at the loc of card button 2
```

You can simply write

```
select card button 2
```

Besides being shorter, this new command lets you select a button or field regardless of what the current tool is.

Simulation with the *drag* Command

The `drag` command lets you simulate mouse drags from scripts with numerical precision. Figure 16-4 shows the syntax for the `drag` command.

```
drag from <loc> to <loc> [with modKey[,modKey2[,modKey3]]]
```

x,y coordinate pairs as number values or in containers

optional commandKey, optionKey, or shiftKey

Figure 16-4. The `drag` command syntax.

Admittedly, the most obvious uses for `drag` are in chapter 17, "Graphics,"

where we turn these simulation commands loose on HyperCard's Painting features. However, there are some nice things you can do with `drag` when manipulating objects, too. We have seen how to make a copy of an object with copy and paste. We have also seen how to reposition an object with `set loc`. The `drag` command lets us do both tasks at the same time.

First, look at all the things you can do with `drag`. If you drag a button by its middle, you reposition it. If you drag it by a corner, you resize it. If you reposition with the Shift key pressed down, the object can only move vertically or horizontally (and, in the case of round rect buttons, the button height is changed to the official Macintosh specifications for round rect buttons). Repositioning with the option key down clones a copy. Dragging with the command key down makes a new, transparent card button. You have similar capabilities when dragging with the field tool. You can simulate each of these actions using the `drag` command when you need to create or manipulate buttons in a script. This gives you the ability to provide a button that will add new buttons, or fields, to a stack. You can also use these techniques to create games with moving buttons. Here is an example that demonstrates object manipulation techniques.

Example 16:

1. Go to the Simulation card of your HyperTalk Bible stack.
2. Make a new card button, place the button just below the Objects menu, and name it `Dragoon`.
3. Enter the following handler in the Dragoon button:

```
on mouseUp -- Dragoon button
  choose button tool
  set dragSpeed to 72 -- 72 makes for a nice gliding drag
  -- clone a copy and put it toward the center of the card
  drag from the loc of me to 250,200 with optionKey
  put item 3 of the rect of me into tempX
  put item 4 of the rect of me into tempY
  -- make original button larger by 20 pixels on a side
  drag from tempX- 2, tempY-2 to tempX + 20, tempY + 20
  -- clone a copy of the bigger button and put it up and to the
  -- left
  drag from the loc of me to 75,100 with optionKey
  -- a constrained move of the top left button to a position
  -- even with but below the original button
  drag from the loc of bkgnd button the number of bkgnd ¬
  buttons to the loc of me with shiftKey
  -- restore the original button to regulation round rect height
     drag from tempX + 18, tempY + 18 to tempX -2, tempY + 18 with ¬
shiftKey
  choose browse tool
end mouseUp
```

4. Click the Dragoon button.

Result:

A clone of the Dragoon button drifts gracefully down the screen. Then, the original button gets bigger. A second clone drifts off to the left side and then

drifts back until it is directly under the original. Finally, the original button reverts to regulation height.

This example shows the techniques you can use to manipulate buttons (or fields). Imagine a card on which the buttons troop in through a graphic door and scurry around to their assigned places. Whether the application is a game, educational software for kids, or a productivity program in which the user is encouraged to rearrange the screen, these object manipulation capabilities can add both functionality and magic to your HyperCard stacks.

The *dragSpeed* Property

Note that in this last example, we used a new property called `dragSpeed`. This is a peculiar global property because it resets itself to 0 on `idle`. This means that if you don't want the default speed, you must set it to the desired value each time you start a script. If you were to comment out the `set dragSpeed` statement in the last example and then run the handler again, the `dragSpeed` would not be 72 but would have reset itself to 0. To make things even more interesting, although a `dragSpeed` of 100 (100 pixels per second) is faster than a `dragSpeed` of 72 (72 pixels or 1 inch per second), the fastest possible `dragSpeed` is 0. We use `dragSpeed` a good deal more in the next chapter.

Constrained *drag* Is Good
If you are creating graphic pictures in HyperCard using simulation of the paint tools, you should be happy to know that you can use the `drag` command with the `commandKey` constant (`drag...with commandKey`) to cause an elongation of a picture in the vertical or horizontal direction and `drag...with commandKey, shiftKey` to enlarge or shrink a picture proportionally.

Summary Scripts

Now that you have seen all the simulation tools, we can cut loose and show some real examples. Simulation is invaluable for automating repetitive tasks, and the following examples demonstrate some new ways to make working with HyperCard even easier. Say you want to automate the creation of several similar buttons. The next handler does this with only a click of a button.

Example 17:

1. Go to the Simulation card of your HyperTalk Bible stack.
2. Make three new card buttons and name them `Make Buttons`, `Move Buttons`, and `Delete Buttons`.

3. Enter the following handler in the Make Buttons button:

```
on mouseUp -- Make Buttons button
  doMenu "New Button"
  put the number of buttons into numButtons
  set rect of button numButtons to 10,40,35,65
  set name of button numButtons to "A"
  put item 1 of the loc of button numButtons into xLoc
  put item 2 of the loc of button numButtons into yLoc
  -- slide determines the horizontal interval between buttons
  put 35 into slide
  -- ascii is the ASCII code for upper case A
  put 66 into ascii
  repeat with n = 1 to 6
    -- drag & clone from center of button to new position
    drag from xLoc,yLoc to xLoc + slide,yLoc with optionKey
    -- name the clone the character equivalent of the ASCII code
    set name of button (numButtons + n) to numToChar(ascii)
    add 35 to slide -- increase slide for next button position
    add 1 to ascii -- increase ascii for the next button name
  end repeat
  choose browse tool
end mouseUp
```

4. Click the Make Buttons button.

Result:

A button appears on the screen. It is repositioned, resized, renamed, and then cloned six times so that you end up with a line of seven similar buttons, as shown in Figure 16-5.

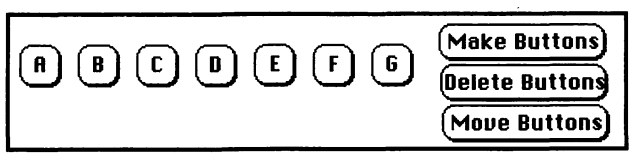


Figure 16-5. Results of example 17.

Note that each of the buttons has a unique name and that they are all neatly spaced across the screen. You can see that this handler could be easily altered to create more buttons in more than one line. The `numToChar()` function is used to convert ASCII numbers (which can be incremented easily by adding 1 to the number each time through the loop) to the letters we want as names for the buttons.

Say that you really meant for the buttons to be background buttons rather than card buttons. In this case, you might want to delete these and start fresh, but if you have already made changes to each of the buttons—added individual scripts perhaps—you might want to move them without losing all your work.

Example 18:

1. Put the following handler into the Move Buttons button:

```
on mouseUp -- Move Buttons button
  choose button tool
  repeat until the number of buttons is 3
    click at the loc of button the number of buttons
    doMenu "Cut Button"
    doMenu "Background"
    doMenu "Paste Button"
    doMenu "Background"
  end repeat
end mouseUp
```

2. Click Move Buttons.

Result:

The buttons are cut one at a time and pasted to the background.

It's even easier to get rid of a gang of buttons.

Example 19:

1. Put the following handler into the Delete Buttons button.

```
on mouseUp -- Delete Buttons button
  choose button tool
  repeat until the number of buttons is 2
    click at the loc of button the number of buttons
    doMenu "Clear Button"
  end repeat
end mouseUp
```

2. Click Delete Button.

Result:

The buttons are deleted one at a time until they are all gone.

Those buttons we made didn't do anything. Let's add a little bit more to our Make Buttons button script, so it loads scripts into the buttons that make them play the notes A through G.

Example 20:

1. Modify the Make Buttons script as follows:

```
on mouseUp
  doMenu "New Button"
  put the number of buttons into numButtons
  set rect of button numButtons to 10,40,35,65
  set name of button numButtons to "A"


     -- new material below --
  put "on mouseUp" & return into tempVar
  put "play" && quote & "boing" & quote && quote & "a" & quote ¬
  & return after tempVar
  put "end mouseUp" after tempVar
  set script of button numButtons to tempVar
```

continued

```
    put script of button numButtons into scriptVar
    -- end new material --

    put item 1 of the loc of button numButtons into xLoc
    put item 2 of the loc of button numButtons into yLoc
    put 35 into slide
    put 66 into ascii
    repeat with n = 1 to 6

      drag from xLoc,yLoc to xLoc + slide,yLoc with optionKey
      set name of button (numButtons + n) to numToChar(ascii)

              -- new material below --
      put "play" && quote & "boing" & quote && quote & ¬
            numToChar(ascii) & quote & return into line 2 of scriptVar
      set script of button (numButtons + n) to scriptVar
      -- end new material --

      add 35 to slide
      add 1 to ascii
    end repeat
    choose browse tool
  end mouseUp
```

2. Click Make Buttons.

Result:

The buttons look just like the previous ones looked, but when you click them they play the appropriate note.

Close variations on these handlers do the same things for fields.

Review Questions

1. Which of the following menu items is unavailable from the Browser user level?
   a. New Card
   b. Compact Stack
   c. Card Info...
   d. all three
2. List the four commands that you use to put a selected field onto the Clipboard.
3. Write a `doMenu` command statement to show the information dialog box for a selected button.
4. Show two ways to select card field ID 2. Use the `click at` command in A and version 1.2's `select` command in B.
5. Write a three-line script fragment (no `on` or `end` keywords), using the `click at` command, that would show the script of button 2.
6. Write a single command statement, using the `edit script` command that shows the script of button 2.

7. What is the fastest way of getting to the script of either a button or a field in version 1.2?
8. Write a `type` command that quits HyperCard.
9. Which of the following would not work as the parameter in a `choose` command?

   a. `buttonTool`
   b. `ButtON tOOl`
   c. `button tool`

10. Which of the following uses of the `tool` function is incorrect?

    a. `if the tool is "browse tool" then beep`
    b. `get tool`
    c. `set the tool to button tool`

11. Are there any field properties you can set as an author but not as a scriptor?
12. Are there any field properties you can set as a scriptor but not as an author?
13. If you want to make sure field 3 is always scrolled to the top when card 1 is opened, what command would you put in a closeCard handler of card 3's script?
14. How would you make a copy of button ID 2 and move it to screen location 300,200 with two commands?
15. The following script fragment makes three buttons: #1, #2, and #3. For each button, tell if it is in the background or card layer.

    a. button 1
    b. button 2
    c. button 3

```
doMenu "New Button" -- makes button 1
doMenu "Copy Button"
doMenu "Background"
doMenu "Paste Button" -- makes button 2
drag from loc of button 1 to 200,50 with optionKey -- button 3
```

16. A script makes the following objects in this order: button ID 1, field ID 1, field ID 2, button ID 2. How many times would you need to repeat the `doMenu "Bring Closer"` command with button ID 1 selected to make button ID 1 the top button?

Exercises

1. Write one button script that opens the Info dialog for the card if the option key is down, for the background if the command key is down, or for the stack if no keys are down.
2. Extend the script from Exercise 2 so that it also shows the scripts of each object if you also hold down the Shift key.
3. Make some buttons to control the automatic scrolling field.

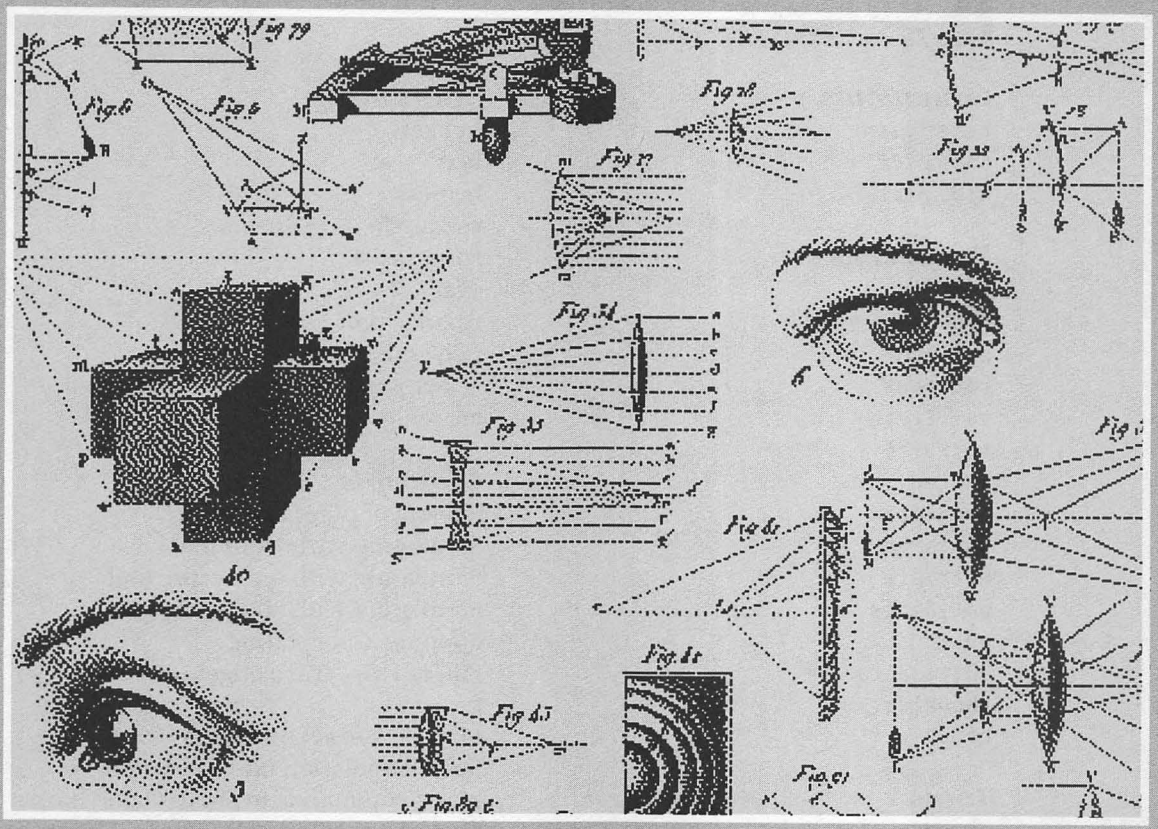
   a. The first button controls the scroll rate and shows the current setting. It lets you increase the scroll number by clicking the button or decrease the

number by clicking with the option key down. Of course, if you hold the mouse down, it should keep adding to or subtracting from the scroll number.

b. The second button should cause the field to scroll up as long as the mouse is held down.
c. The third button should cause the field to scroll down as long as the mouse is held down.
d. The fourth button should be as extension of item b that causes the field to scroll up from wherever it was to the end.
e. The last button should be an extension of item c that causes the field to scroll down from wherever it was to the top.

Chapter 17

Graphics



Quick View

Commands
`reset paint`
`show pict`
`hide pict`

Properties
`lineSize`
`brush`
`pattern`
`textFont`
`textStyle`
`textSize`
`filled`
`dragSpeed`
`centered`
`multiple`
`polySides`
`grid`
`multispace`
`showPict`
`hidePict`

Tools
pencil
brush
spray

Tools *cont'd*
bucket
text
browse
rectangle
round rect
oval
regular polygon
polygon
select
eraser

Concepts
Review of paint concepts
Simulating with Paint menu
Simulating with applicator tools
Simulating with shape tools
Flipping and rotating
Fill, darken, trace edges, etc.
Lasso tool
`doMenu "Select"` to lasso
Some simulation odds and ends
Menu commands to show dialog boxes
Showing and hiding pictures (Version 1.2)
Random art script

Chapter 17

Graphics

Almost everything you can do with HyperCard's painting tools you can simulate in a script with HyperTalk. In addition, you can write scripts that create mathematically precise designs that you could never do by hand. To do these things, you need a handful of commands, such as `choose`, `doMenu`, `type`, `click`, `drag`, and `set`, as well as a working knowledge of the painting tools, the painting properties, and the Painting menu commands. The commands were discussed in chapter 16, "Simulations", and if you are unfamiliar with them you might want to review that chapter before you attempt this one. The tools, properties, and menu items for doing graphic simulation are covered in this chapter.

Before we go into detail, however, we should say a word or two about painting in general. You will find graphic simulation much easier if you thoroughly understand what happens when graphic images appear on the screen. HyperCard's painting environment is an extension, and improvement, on the popular Macintosh MacPaint environment. All images are bit-mapped, as they are in MacPaint, and thus exist as a collection of pixels on the screen. With HyperCard, the situation is complicated by the existence of two graphic layers on each card.

The top layer is the card graphic layer, which is unique to the one card (unless you copy the graphic or the entire card). Below the card graphic layer lies the background layer, which is common to all cards sharing the same background. Think of the background layer as a white paper surface and the card graphic layer as a clear plastic overlay sheet. You can apply paint to either layer. Dual layers can cause some problems when both layers are blank, because it is hard to tell which layer you are drawing in. In HyperCard versions 1.0.1 through 1.2.2, you need worry about only two colors: black and white. You can use the painting tools to apply black and white to either the card overlay sheet or the background paper. Keep in mind that applying either black or white to the card overlay conceals the background paper and the background buttons and fields. If you have some black pixels in the background and apply white on the card above them, the pixels below disappear, but they are really still there in the background.

Because this is electronic paint we are applying, HyperCard gives us the ability to turn transparent any area of white paint by selecting the Transparent menu item.

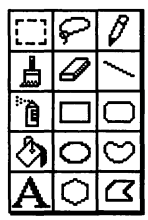
You are less likely to conceal background graphic images by accident if you

understand all the ways such graphics can be concealed. Opaque buttons and fields in the card or background levels can block the view, as can white or black paint on the card graphic layer.

Mac II and a Visual Effects Bummer
If the user of your stack is viewing it on a Macintosh II computer, the following steps are necessary for the visual effects in HyperCard to work properly. The screen of the Mac II must be set to 2 grays (1-bit depth) in the Control Panel's monitor menu. Move HyperCard's windows to the main screen (the one with the menu bar—Mac IIs can support more than one monitor) and make sure that HyperCard's window doesn't cross multiple monitors (like the Radius Two Page Display allows).

Simulating with the *choose* Command

The `choose` command accepts as its parameter one of the 18 Tool Palette tools. We covered the first three tools, the general tools, in the last chapter. In this chapter, we cover the 15 painting tools (as shown in Figure 17-1). One or another of these tools must be active for the Painting menu items to be available. In short, a painting tool must be chosen before you can do anything to the graphic layers of your stacks. Every example in this chapter begins with a command that chooses a tool. The only question is which tool you should choose.



Legend:		
select tool	lasso tool	pencil tool
brush tool	eraser tool	line tool
spray tool	rectangle tool	round rect tool
bucket tool	oval tool	curve tool
text tool	regular polygon tool	polygon tool

Figure 17-1. The painting tools.

In addition to painting tools, the painting properties are an integral part of simulating graphics. Setting properties is the equivalent of making selections in the Text Style, Brush Shape, and all the other Painting dialog boxes. As we discuss the tools, we also introduce the properties. See the following list of the painting properties and their values.

Painting Properties

brush	1 to 32
centered	true, false
filled	true, false

grid	true, false
lineSize	1, 2, 3, 4, 6, 8
multiple	true, false
multiSpace	1 to 9
pattern	1 to 40
polySides	3, 4, 5 6, 8, 0
textAlign	left, center, right
textFont	varies with system
textHeight	any number
textSize	varies with font
textStyle	bold, italic, outline, shadow, extend, underline, condense, plain
dragSpeed	1 to 1000 to ?
editBkgnd	true, false

Simulating the Applicator Tools

The first group of painting tools we examine are the ones used to apply white or black paint to the screen in lines, dots, or by pouring the paint on the screen. We call these applicator tools, and they share many of the same characteristics of the tools for making shapes and borders (shape tools), which are covered in a later section of this chapter.

The Pencil Tool

The pencil tool is the simplest of the tools you can use to make graphic images. You can use it to make simple, one-pixel-wide, white or black lines. You can also use it to make single pixel dots. When you drag with the pencil, HyperCard knows whether the starting point of the drag was over a white or a black pixel and draws a one pixel line of the other color—white lines come from black areas and black lines from white areas. Dragging the pencil tool with `shiftKey` constrains the line so that it is either horizontal or vertical (`drag from 10,10 to 100,100 with shiftKey`). The advantage of using the pencil over other tools, like the brush and the line, is that the pencil is always a one-pixel black or white line, whereas the brush can be any shape, and the line can use varying line sizes. This means that you can use the pencil tool without having to set any properties. The disadvantage of using the pencil is that it is not easy to know in advance whether a line will be drawn black or white unless you know the color of the pixel on which the drag starts. Even if you keep track of every black pixel in the card layer, there is a chance you will start your pencil drag on a transparent card layer pixel over a black background pixel, which results in a white line. Although you will most often use the pencil tool to draw lines, it can also be used to draw individual dots on the screen.

Example 1:

1. Open the last card of your HyperTalk Bible stack, make a new card, and name it `Graphics`.
2. Make a new card button and name it `Pencil Drag`.
3. Enter the following handler in the Pencil Drag button:

```
on mouseUp -- Pencil Drag button
  reset paint -- line 2
  choose pencil tool
  drag from 100,100 to 150,50
  drag from 151,50 to 201,100
  drag from 201,101 to 151,151
  drag from 150,151 to 100,101
end mouseUp
```

4. Click the Pencil Drag button.

Result:

Draws a diamond shape.

Note that the starting point for each drag is slightly off the end point of the preceding drag; for example, the first line ends at 150,50 and the next line starts one pixel to the right at 151,50. If the next drag started on the black pixel at the end of the previous drag, a white line is created.

`reset paint`, in line two of the example, is a simple command that restores all the paint properties to their default values. We include it here to make sure your properties agree with ours so that we can predict your results.

The Line Tool

The line tool draws a black line of the current `lineSize`. You can use the line tool to draw lines and dots just as you did with the pencil tool, but with the line tool, the line thickness can be changed. You don't have to worry about accidental white lines with the line tool because it always draws in black. The line can be 1, 2, 3, 4, 6, or 8 pixels wide and is set with the `lineSize` property. Dragging `with shiftKey` constrains the line angle to 15° increments, which can come in handy if you want to draw lines that are not too jagged without bothering to calculate the exact coordinates. Dragging `with optionKey` causes the line to be drawn using the current pattern, so you actually can draw in white if you wish—but not accidentally.

Example 2:

1. Open the Graphics card in your HyperTalk Bible stack.
2. Add a new card button to the HyperTalk Bible stack and name it `Line Drag`.
3. Enter the following handler in the Line Drag button.

```
on mouseUp -- Line Drag button
     choose line tool -- line size 1
  hide menuBar
     set lineSize to 1
```

continued

```
        put 100/5 into tempVar -- line 5
        repeat with n = 1 to 5
             drag from 1,tempVar*n to 100,1
             drag from 100-(tempVar*n),100 to 100,1
             drag from 1,1 to 100,tempVar*n
             drag from 1,1 to tempVar*n,100
        end repeat
end mouseUp
```

4. Click the Line Drag button.

Result:

You should get an image identical to Figure 17-2.

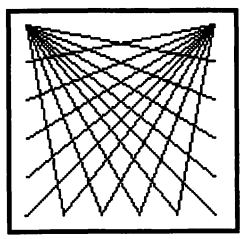


Figure 17-2. Example 2 vanishing point drawing with line tool.

In this example, we use the `set` command with the `lineSize` property to guarantee that your lines look like the ones in Figure 17-2, regardless of the `lineSize` setting before the script ran. Setting the `lineSize` to 1 draws a 1-pixel line. Line five in the script produces an interval so that you can draw evenly spaced lines along both sides and the bottom of the frame. The second `draw` command moves from right to left across the bottom of the frame (by subtracting an even larger number from 100). The fourth `draw` command moves from left to right. To get the Menu bar back, press Command-Space.

The Brush Tool

The brush tool is another line or dot drawing tool, but it is much more flexible than the pencil or line. The brush paints a line of the current fill pattern. In addition, it uses the current brush shape. There are 32 brush shapes and 40 standard patterns to choose from. In addition to dragging lines, you can also use the brush tool with the `click` command to put one or a thousand impressions of the current brush shape on the screen. You can also use the brush as an eraser by dragging `with commandKey`.

To show you the different effects possible with brush, the next example is a variation on Example 1, using different brush shapes. Note that you no longer have to offset the starting point of one line from the ending point of the previous line because the brush tool is not sensitive to the background color.

Example 3:

1. Open the Graphics card in your HyperTalk Bible stack.
2. Add a new card button to your HyperTalk Bible stack and name it `Brush Shape`.
3. Enter the following handler in the button:

```
on mouseUp -- Brush Shape button
  choose brush tool
  set brush to 8
  set pattern to 12
  drag from 100,100 to 150,50
  drag from 150,50 to 200,100
  drag from 200,100 to 150,150
  drag from 150,150 to 100,100
end mouseUp
```

4. Add another card button and name it `Clear`.
5. Enter the following handler in the button:

```
on mouseUp -- Clear button
  choose select tool
  doMenu "Select All"
  doMenu "Clear Picture"
  choose browse tool
end mouseUp
```

6. Click the Clear button to remove the previous examples and then click the Brush Shape button.

Result:

You should get an image identical to the image shown in Figure 17-3A. Figure 17-3B and Figure 17-3C show the image with some variations in brush shape.

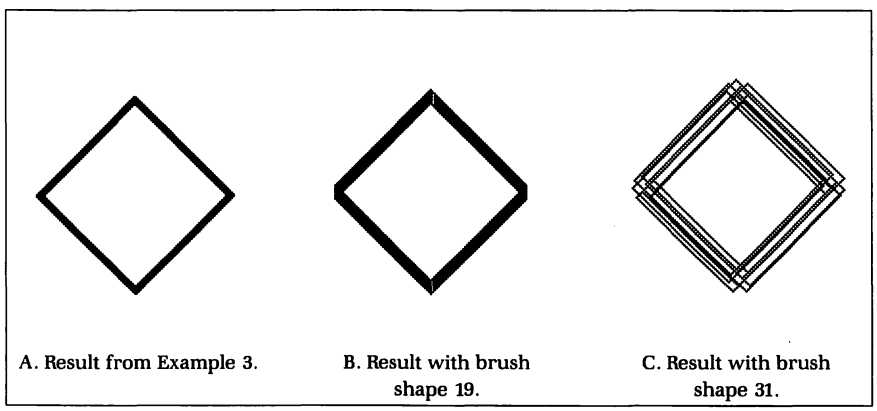


Figure 17-3. Drawings and variations produced by Example 3.

We've used two more properties in this example. `set brush` determines the brush shape, and `set pattern` determines the pattern. Figure 17-3B shows the results of the same handler with the brush shape property set to 19. Figure 17-3C

shows the results with a brush shape of 31. To get different effects, substitute other brush shapes or change the pattern with a `set` command.

We included the Clear button script here because it does a job we need to do repeatedly in this chapter. Basically, it uses the menu commands Select All and Clear Picture to erase all graphics in the current graphic layer. We say more about how this works later in the chapter. Note that this handler chooses the browse tool before it ends. It is a good idea to leave things as close to the way you found them as possible when you end a script. By choosing the browse tool, we save the user the trouble of doing this from the Tool Palette.

The next example demonstrates using the brush tool with the `click` command to plot dots on the screen. See chapter 7, "Mouse," for the syntax of the `click` command. This is a good example of something you can do easily with a language but can't do at all with a painting program. To make the example more interesting, we used the `random()` function to change the brush shape on each repeat.

Example 4:

1. Open the Graphics card in your HyperTalk Bible stack.
2. Add a new card button to your HyperTalk Bible stack and name it `Brush Dots`.
3. Enter the following handler in the button:

```
on mouseUp -- Brush Dots button
  choose brush tool
  repeat with n = 1 to 300
    set brush to random(32)
    put trunc(sin(n) * 100) into x -- line 5
    if x>0 then put x + 200 into x else put 200-x into x
    click at x,n
    click at n,x
  end repeat
  choose browse tool
end mouseUp
```

4. Clear the image created by the last example and then click the Brush Dots button.

Result:

Figure 17-4 shows an image of a portion of the resulting pattern of dots.

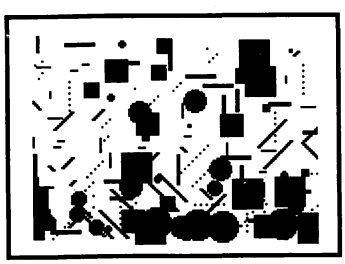


Figure 17-4. Example 4 result.

A few words of explanation about this script. The fifth line gets the sine of the counter variable `n` (times one hundred) and then truncates it to a whole number. The next line adjusts the number so that the action stays on the screen. (This is called clipping.) The `click` commands repeat the same pattern on different axes. Change the formulas to see what new patterns you can produce.

There is no limit to the customized screen patterns you can create with this or similar techniques. You can also add a pattern randomizer for really unusual effects.

The Spray Tool

The spray tool is very similar to the brush tool with the brush shape fixed to shape number 31. However, `drag` seems to work differently with the spray tool. It is essential to set the `dragSpeed` property before dragging with the spray tool. If you don't, you get what looks like one click at the start of the drag and one click at the end of the drag and nothing in between. The best drag speeds seem to be between 30 and 500 with the slower drags producing a more interesting effect. Everything said previously about brush applies to this tool as well.

Example 5:

1. Open the Graphics card in your HyperTalk Bible stack.
2. Add a new card button to your HyperTalk Bible stack and name it `Graffiti`.
3. Enter the following handler in the button:

```
on mouseUp -- Graffiti button
  choose spray tool
  hide menuBar
  set dragSpeed to 100
  set pattern to 12
  drag from 20,100 to 60,20
  drag from 60,20 to 100,100
  drag from 100,100 to 140,20
  drag from 140,20 to 180,100
  click at 200,100
  drag from 240,20 to 240,100
  drag from 240,100 to 340,100
  click at 360,100
  drag from 200,120 to 200,200
  drag from 100,160 to 300,160
  drag from 20,220 to 180,220
  drag from 100,220 to 100,300
  click at 200,300
  drag from 240,220 to 340,220
  drag from 340,220 to 240,300
  drag from 240,300 to 340,300
  click at 360,300
  choose browse tool
end mouseUp
```

4. Press the Clear button and then click the Graffiti button.

Result:

You should get an image identical to Figure 17-5.

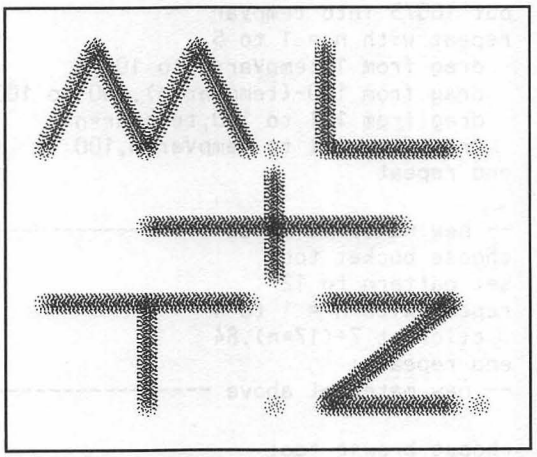


Figure 17-5. Example 5 spray tool result.

Although you can't vary the spray tool dot pattern, you can change the paint pattern. You can use the spray tool to simulate very neat graffiti, and it may be the perfect answer to boring borders.

The Bucket Tool

When you want to pour a paint pattern inside a shape or all over a screen, use the bucket tool. Clicking the bucket tool from a script causes the current pattern to be poured on the screen starting at the coordinates specified. If the pixel at that location is white, the pattern fills all the white area until a black border is reached. If the pixel is black, all the black is filled until a white border is reached. In either case, a gap of even one pixel in a border lets the pattern leak out. You can use the bucket tool to fill anything with the pattern, even text, but be careful or you will end up with a screen full of your pattern.

The next example uses the bucket tool for precise paint pouring. This should demonstrate how useful this tool can be and also how careful you must be when you use it. If your coordinates are off, you can easily ruin a graphic.

Example 6:

1. Open the Graphics card in your HyperTalk Bible stack.
2. Make a copy of your Line Drag button; rename the copy `Careful Fill` and edit the handler so that it looks like the following:

```
on mouseUp -- Careful Fill button
  choose line tool -- line size 1
  hide menuBar
     set lineSize to 1
  put 100/5 into tempVar
  repeat with n = 1 to 5
    drag from 1,tempVar*n to 100,1
    drag from 100-(tempVar*n),100 to 100,1
    drag from 1,1 to 100,tempVar*n
    drag from 1,1 to tempVar*n,100
  end repeat

  -- new material below ----------------
  choose bucket tool
  set pattern to 12
  repeat with n = 1 to 4
    click at 7+(17*n),84
  end repeat
  -- new material above ----------------

  choose browse tool
end mouseUp
```

3. Press the Clear button and then click the Careful Fill button.

Result:

You should get an image identical to Figure 17-6.

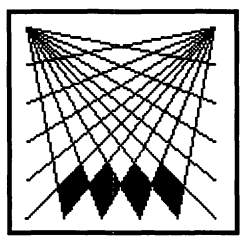


Figure 17-6. Example 6 vanishing point drawing after the bucket tool.

I suppose we could call this example Charley Brown's shirt in perspective.

The Text Tool

The text tool can be used in conjunction with the `click` and `type` commands to put graphic (bit-mapped, noneditable) text in either the card or background graphic layer. Typing text to the screen has a slightly uncanny look. Until you can teach your computer to talk like HAL in the movie *2001,* this is probably the best way to make your computer seem interactive.

Example 7:

1. Open the Graphics card in your HyperTalk Bible stack.
2. Add a new card button to your HyperTalk Bible stack and name it `Text`.
3. Enter the following handler in the button.

```
on mouseUp -- Text button
  choose text tool
  click at 50,50
  type "I'm sorry, Dave... but I can't do that."
  choose browse tool
end mouseUp
```

4. Click the Clear button and then click the Text button.

Result:

The letters in the type string appear slowly on the screen as if someone were typing them.

So far so good, but what if you want something other than the default text attributes? There are five properties that control how text looks, and you can set them all from your scripts. The text properties and their possible values are shown in the list titled "Painting Properties." `textAlign` determines whether text is centered, right justified, or left justified. `textFont` sets the font. `textHeight` determines the distance between lines. `textSize` sets the size of the text; and `textStyle` lets you italicize, underline, outline, etc., your text. You can combine styles in the same command to give bold, outlined, and extended text.

You are probably accustomed to cumulative text style changes in which selecting bold, for instance, adds that attribute to any other style attributes already selected and selecting bold again removes the attribute. HyperTalk's `textStyle` property works differently and each `set textStyle` command replaces all existing attributes with the ones in the new command.

If you don't have the New York font in your system, you can substitute any other font and get similar results. Unfortunately, in the middle of typing, you cannot simulate pressing the Enter key to "fix" the attributes of the existing text (as you can in MacPaint and HyperCard). This normally lets you change the attributes of subsequent typing without also changing the attributes of the text that preceded the Enter keypress. In a script, you have to use the `click` command each time you want to change the text properties of only subsequent typing.

Example 8:

1. Open the Graphics card in your HyperTalk Bible stack.
2. Add a new card button to your HyperTalk Bible stack and name it `Crazy Text`.
3. Enter the following handler in the button.

```
on mouseUp -- Crazy Text button
  choose text tool
  -- substitute for any fonts that you don't have
  put "Chicago,Geneva,Monaco,New York" into fonts
  put "bold, outline,underline,shadow,italic" into styles
  put "9,10,12,18" into sizes
```

continued

```
    put "Press mouse button to end the show." into msg
    repeat until the mouse is down
      click at random(400),random(300)
      set textFont to item random(4) of fonts
      set textStyle to item random(4) of styles
      set textSize to item random(4) of sizes
      type "HyperTalk For Everyone"
    end repeat
    choose browse tool
  end mouseUp
```

4. Click the Clear button and then click the Crazy Text button.

Result:

The string is printed at random locations on the screen in a variety of fonts, sizes, and styles.

Substitute for any fonts you don't have in your system. This script first stores a selection of possible property values for each of the text properties, separated by commas, in the local variables fonts, sizes, and styles. Then, it uses the `set` command and the `random()` function to grab different combinations of these values and apply them to the message string.

Simulating the Shape Tools

The shape tools in the Painting Palette are like the applicator tools, but the shape tools produce geometric shapes that can also be used for making borders. For the most part, all the shape tools work about the same way.

The Rectangle Tool

The rectangle tool can be used either to make borders surrounding other graphic elements or simply to make rectangle shapes that can then be combined to form complex structures. The rectangle is black unless you `drag with optionKey`, in which case the tool uses the current pattern for its outline. The width of the rectangle border is determined by the current `lineSize`. Dragging `with shiftKey` constrains the rectangle to a square.

When `filled` is true, the rectangle, or any other shape, is drawn with its insides filled with the current paint pattern. Dragging a rectangle `with optionKey` and with `filled` set to true creates a rectangle with the center and the border in the current pattern.

Example 9:

1. Open the Graphics card in your HyperTalk Bible stack.
2. Add a new card button to your HyperTalk Bible stack and name it `FilledRectangles`.
3. Enter the following handler in the button.

```
on mouseUp -- Filled Rectangles button
  choose rectangle tool
  set filled to true
  set dragSpeed to 500
  repeat for 30
    set pattern to random(39) + 1 -- line 6
    set lineSize to random(4)
    put random(400) into xVar
    put random(250) into yVar
    put random(100) into wideVar
    put random(100) into highVar
    drag from xVar,yVar to xVar + wideVar,yVar + highVar
  end repeat
  choose browse tool
end mouseUp
```

4. Click the Clear button and then click the Filled Rectangles button.

Result:

A mass of randomly placed, sized, and filled rectangles are drawn on the screen, as shown in Figure 17-7.

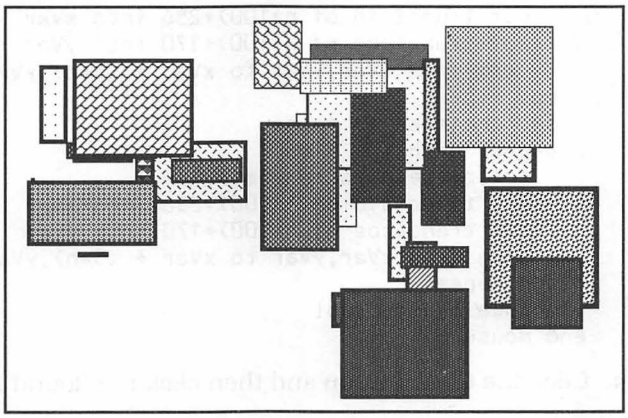


Figure 17-7. Random rectangles from Example 9.

Just to make things interesting in this example, the `lineSize` of the rectangle borders vary, and we set `dragSpeed` to a speed that is not too slow but more interesting to watch than the default speed. Also note that in line 6, to avoid selecting the white pattern, we have shifted the random range up by one so that only the patterns between 1 and 40 can be selected.

The Round Rect Tool and the centered *Property*

The round rect tool is identical to the rectangle tool except that the corners are rounded. Another property used with shapes is `centered`. If `centered`

is true, rectangles are drawn starting from the center rather than from the corner, As a result, a round rectangle drawn with the command `drag 100,100 to 200,200` when `centered` is true is four times the size of a round rectangle drawn with the same command when `centered` is false. Because of this difference, if you are in doubt as to the current setting, test the waters before you start dragging. You could test `centered` with a line like `if centered is true then set centered to false`, but it is quicker to just `set centered to false` whether it was true or false.

Example 10:

1. Open the Graphics card in your HyperTalk Bible stack.
2. Add a new card button to your HyperTalk Bible stack and name it `Round Rects`.
3. Enter the following handler in the button.

```
on mouseUp -- Round Rects button
  choose round rect tool
  set dragSpeed to 500
  set filled to true
  set lineSize to 1
  set centered to true
  repeat with n = 1 to 7
    set pattern to random(39) + 1
    put trunc(sin of n*100)+256 into xVar
    put trunc(cos of n*100)+170 into yVar
    drag from xVar,yVar to xVar + (5*n),yVar+(5*n)
  end repeat
  set centered to false
  repeat with n = 1 to 7
    set pattern to random(39) + 1
    put trunc(sin of n*100)+256 into xVar
    put trunc(cos of n*100)+170 into yVar
    drag from xVar,yVar to xVar + (5*n),yVar+(5*n)
  end repeat
  choose browse tool
end mouseUp
```

4. Click the Clear button and then click the Round Rects button.

Result:

Because the first batch of round rectangles are drawn with the centered property true, whereas the second batch, superimposed over the first, is drawn with the centered property false, you see an overlapping spiral effect, as shown in Figure 17-8.

The Oval (Circle) Tool

The oval tool is also similar to the rectangle tool except that, in this case, if you drag with the Shift key the shape is constrained to a circle. Thus, it's really a circle and oval tool.

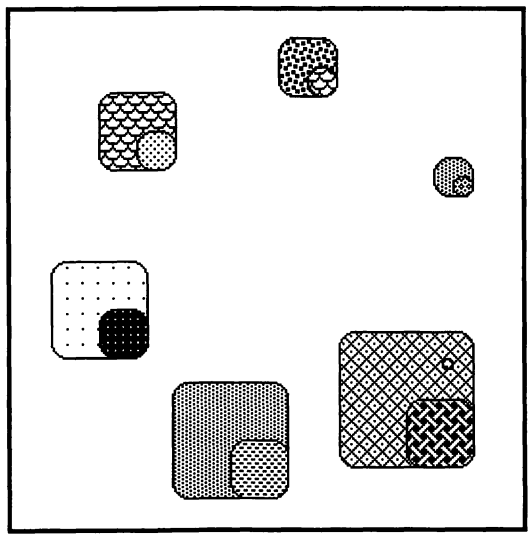


Figure 17-8. Round rectangles from Example 10.

Example 11:

1. Open the Graphics card in your HyperTalk Bible stack.
2. Add a new card button to your HyperTalk Bible stack and name it `Oval`.
3. Enter the following handler in the button.

```
on mouseUp -- Oval button
  choose oval tool
  set filled to false
  set centered to true
  repeat with n = 1 to 30
    drag from 250,170 to 250 + (n * 3),170 + (n * 3)
  end repeat
  choose browse tool
end mouseUp
```

4. Click the Clear button and then click the Oval button.

Result:

The `repeat` loop draws a dense collection of concentric circles. (See Figure 17-9.)

There is a shape property called `multiple` that lets you draw multiple shapes with a single drag. In the next example, we use this property to make, with a single drag, a design similar to the one made in the last example.

Example 12:

1. Open the Graphics card in your HyperTalk Bible stack.
2. Add a new card button to your HyperTalk Bible stack and name it `Easy Ovals`.
3. Enter the following handler in the button.

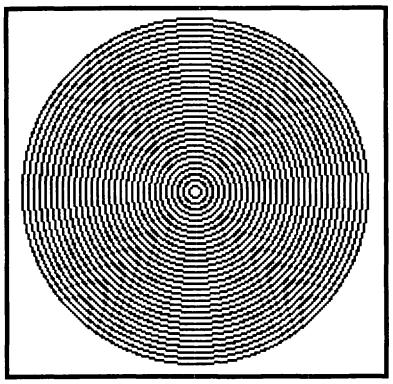


Figure 17-9. Example 11's concentric circles.

```
on mouseUp -- Easy Ovals button
  choose oval tool
  set centered to true
  set multiple to true
  set dragSpeed to 100
  drag from 250,170 to 340,260
  choose browse tool
end mouseUp
```

4. Click the Clear button and then click the Easy Ovals button.

Result:

Concentric circles, irregularly spaced (see Figure 17-10).

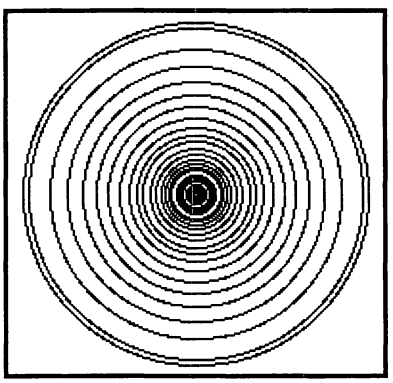


Figure 17-10. Example 12's concentric circles with `multiple`.

For different effects, you can try the same script with different `dragSpeeds`. Each speed creates a different picture because slower speeds have more time to draw more circles, but faster speeds run out of time and have to skip some possible circles. Although this example is shorter than Example 11, the effect is not quite the same. If you want the identical results, you have to take advantage of yet another property, `multiSpace`, and adjust the `dragSpeed`. Add the following two lines to Example 12 to duplicate the concentric circles in Figure 17-9:

```
set dragSpeed to 20 -- these lines replace the set dragSpeed line
set multiSpace to 3
```

In this case, the secret is the combination of the `multiSpace` and `dragSpeed` properties. When the `multiSpace` property is set to 3, the interval between circles is limited to a minimum of 3 pixels, which was the interval used in Example 11. The `dragSpeed` is very slow so that every possible line is actually drawn. You might be able to use a faster `dragSpeed` if you are using a Mac II or a Mac with an accelerator board, but on a Mac Plus or an SE, a faster `dragSpeed` doesn't allow time to draw all the circles.

The Curve Tool

The curve tool, a strange device under the best of circumstances, can't be used at all for simulation.

The Regular Polygon Tool

The HyperCard regular polygon tool creates polygons with regular sides and always draws from the center regardless of the centered property setting. A regular polygon is a closed figure with 3, 4, 5, 6, 8, or 0 (a circle) sides. Why there isn't a seven-sided polygon is a mystery, but we can live with this lapse. The number of sides a polygon will have is determined by the `polySides` property. As with the other shapes, the polygon is drawn with the current `lineSize` in black or if you drag with `optionKey`, in the current pattern.

Dragging with `shiftKey` constrains the orientation of the polygon to 15° increments. If you have played with this tool in Painting mode, you know that you can produce interesting effects by rotating a regular polygon around its center point while the mouse button is pressed down. The mouse position when you release the mouse button determines the orientation of the shape. It would be nice to be able to simulate turning a regular polygon on its axis. Unfortunately, this is not possible in HyperTalk because script drags can only be straight lines.

As with the other shapes, regular polygons are most frequently used as geometric elements in a picture or to create borders for other graphic elements on a screen.

The Polygon Tool Can't Be Controlled
All the paint tools, except the polygon tool, can be controlled from a script in HyperTalk. The polygon tool doesn't work with drag. If you need to play with irregular shapes, use the curve tool or the line tool. Note also that a lassoed object can be dragged only in a straight line.

Example 13:

1. Open the Graphics card in your HyperTalk Bible stack.
2. Make a new card button in your HyperTalk Bible stack and name it `Polygons`.
3. Enter the following handler in the button:

```
on mouseUp -- Polygons button
  choose regular polygon tool
  set centered to true
  put "3,4,5,6,8" into sides -- line 4
  repeat with n = 1 to 5
    set polySides to item n of sides -- line 6
    drag from 70 * n,170 to 70 * n,200
  end repeat
  choose browse tool
end mouseUp
```

4. Click the Clear button and then click the Polygons button.

Result:
All the polygon shapes you can set with the `polySides` property are dragged out. See Figure 17-11.

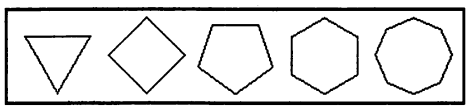


Figure 17-11. Random rectangles from Example 13.

Quiz: What's missing in Figure 17-11? If you said the 0-sided polygon you are correct. Versions 1.0.1 through 1.2 of HyperCard don't let you set `polySides` to 0, so you can only draw the circle with the regular polygon tool if you choose 0 manually in the Polygon Sides dialog box. In this example, we put the possible `polySides` values into a variable called `sides` in line four and then pulled those values out one at a time, using the incrementing value of the variable `n`, in line six.

The Polygon Tool and the grid *Property*

This tool, at least in version 1.0.1 through 1.2, can't be simulated in a script. You probably won't be using the `grid` property either, but it is there if you need it. The snap grid is like a built-in rounding function that takes a mouse click

anywhere in an 8-pixel by 8-pixel square and converts it to a mouse click at the center of that square. When you are drawing free hand, this gives you a high level of precision and regularity. A snap grid isn't as useful when you are drawing with numbers (screen coordinates), which is what simulation is all about, but if you want to be sure you are covering every eventuality in your scripts, you could set grid to false at the beginning to be on the safe side.

Simulating the Manipulator Tools

The last group of tools are used to move, erase, stretch, or otherwise change images created with the previous tools. We also introduce you to some of the menu commands from the Paint menu, which are used with the `doMenu` command to select or to manipulate selections. See Figure 17-12 for a complete list of these commands.

File	Edit	Paint	Options
Import Paint...	Undo Z	Select S	Grid
Export Paint...	Cut Picture H	Select AM A	FatBits
Quit HyperCard Q	Copy Picture C	Fill	Power Keys
	Paste Picture V	Invert	Line Size...
	Clear Picture	Pickup	Brush Shape...
	Text Style... T	Darken	Edit Pattern...
	Background B	Lighten	Polygon Sides...
		Trace Edges	Draw Filled
		Rotate Left	Draw Centered
		Rotate Right	Draw Multiple
		Flip Vertical	
		Flip Horizontal	
		Opaque	
		Transparent	
		Keep K	
		Revert	

Figure 17-12. Painting menu commands.

The Select Tool

The select tool is used to select an area of a card or background graphic layer. The select tool is like the MacPaint selection rectangle, but it also has a lasso option: If you drag across the screen with `optionKey`, only the black pixels inside the section rectangle are selected, and the tool changes to the lasso tool. If you `drag... with commandKey`, the selecting rectangle shrinks to fit the black graphic

elements inside the selection, creating the smallest possible rectangle and eliminating any extraneous white space. For simulation purposes, you almost always use one of the versions of the select tool when you need to select an area of the screen. (There are problems with lasso that we explain in a moment).

Once an area has been selected, it can be moved, copied, cloned, pasted, cut, stretched, or cleared. Moving, stretching, and cloning are select tool options. You move a selection by dragging it. Dragging from a corner of the selection `with commandKey` stretches the selection and dragging `with optionKey` causes a clone of the original to be pulled off. Copying, cutting, pasting, and clearing are done with the `doMenu "Copy Picture"`, `doMenu "Cut Picture"`, `doMenu "Paste Picture"`, and `doMenu "Clear Picture"` commands. Remember that if the selection was created by dragging `with optionKey` the tool has changed to the lasso tool, and the lasso manipulation rules (explained later) apply. The next example shows the difference between the different kinds of selections with the aid of the `doMenu "Invert"` menu command. The `doMenu "Invert"` command changes every white (or transparent on the card graphic layer) pixel in a selection to black and every black pixel to white.

Example 14:

1. Open the Graphics card in your HyperTalk Bible stack.
2. Add a new card button to your HyperTalk Bible stack and name it `Select`.
3. Enter the following handler in the button.

```
on mouseUp -- Select button
  choose regular polygon tool
  set multiple to true
  set multiSpace to 2
  set lineSize to 2
  set dragSpeed to 100
  set polySides to 5
  drag from 80,100 to 80,130
  -- above creates original image
  choose select tool
  set centered to false
  set dragSpeed to 1000
  drag from 40,60 to 120,140 -- simple selection
  drag from 80,100 to 170,100 with optionKey -- move and clone
  doMenu "Invert"
  drag from 40,60 to 120,140 with commandKey -- tight selection
  drag from 80,100 to 260,100 with optionKey -- move and clone
  doMenu "Invert"
  drag from 40,60 to 120,140 with optionKey -- lasso selection
  drag from 80,100 to 350,100 with optionKey -- move and clone
  doMenu "Invert"
  choose rectangle tool
  set multiple to false
  drag from 35,55 to 395,145 -- draw border
  choose browse tool
end mouseUp
```

4. Click the Clear button and then click the Select button.

Result:

You should get an image very similar to Figure 17-13A-D.

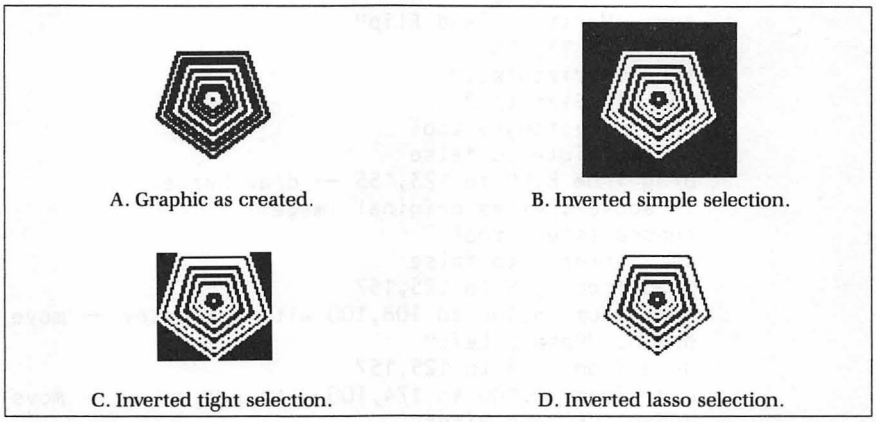


Figure 17-13. Result of Example 14.

This script uses many of the techniques you learned in the shape tools section and combines them with the select tool to create a fairly complex graphic image—it is even interesting to watch.

Flipping and Rotating

The next example uses techniques you've learned in this chapter. These techniques are used to build a demonstration of several new menu commands. This example is very straight-forward as far as the scripting is concerned, but it should give you a good idea of how powerful these tools can be when used for graphic simulation. You can create complex graphic images with text and move them about and change them in ways that will amaze your friends and maybe even yourself.

Example 15:

1. Open the Graphics card in your HyperTalk Bible stack.
2. Add a new card button to your HyperTalk Bible stack and name it `Select Flip`.
3. Enter the following handler in the button.

```
on mouseUp -- Select Flip button
  reset paint
  set dragSpeed to 400
  set textSize to 9
  choose text tool
  click at 13,100
  type "This is a demonstration"
  click at 13,110
  type "of the use of the menu"
  click at 13,120
```

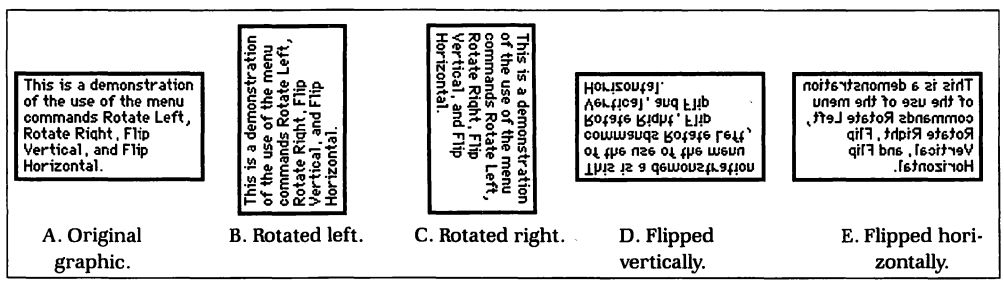
continued

```
        type "commands Rotate Left,"
        click at 13,130
        type "Rotate Right, Flip "
        click at 13,140
        type "Vertical, and Flip"
        click at 13,150
        type "Horizontal."
        set lineSize to 2
        choose rectangle tool
        set multiple to false
        drag from 8,90 to 123,155 -- draw border
        -- above creates original image
        choose select tool
        set centered to false
        drag from 6,88 to 125,157
        drag from 13,100 to 108,100 with optionKey -- move clone
        doMenu "Rotate Left"
        drag from 6,88 to 125,157
        drag from 10,100 to 174,100 with optionKey -- move clone
        doMenu "Rotate Right"
        drag from 6,88 to 125,157
        drag from 10,100 to 270,100 with optionKey -- move clone
        doMenu "Flip Vertical"
        drag from 6,88 to 125,157
        drag from 10,100 to 389,100 with optionKey -- move clone
        doMenu "Flip Horizontal"
        choose rectangle tool
        set centered to true
        drag from 255,122 to 509,188 -- draw border
        choose browse tool
    end mouseUp
```

4. Click the Clear button and then click the Select Flip button.

Result:

See Figure 17-14A-E.



A. Original graphic. B. Rotated left. C. Rotated right. D. Flipped vertically. E. Flipped horizontally.

Figure 17-14. A normal graphic.

You can use the flip and rotate menu commands to create quite a bit of on-screen action. Selection animation just requires a little imagination and a few friendly commands.

Other Paint Menu Items: Fill, Darken, Lighten, Trace Edges, Transparent

The menu items are mostly interchangeable. Rather than `doMenu "Flip Vertical"` you could substitute `doMenu "Fill"` to fill the entire selection with the current pattern (this works best with lasso selections), `doMenu "Darken"` to scatter black pixels inside the selection, `doMenu "Lighten"` to scatter white pixels, or `doMenu "Trace Edges"` to outline the black areas of the selection.

One nice effect you can get with `doMenu "Fill"` is patterned text.

Example 16:

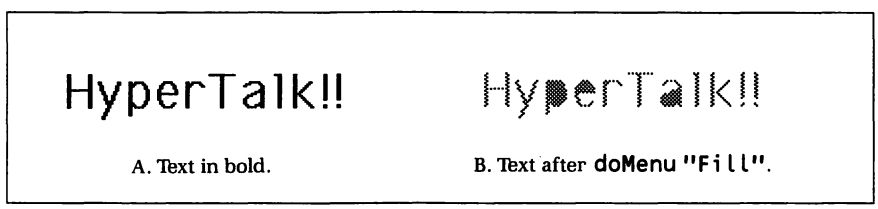
1. Open the Graphics card in your HyperTalk Bible stack.
2. Add a new card button to the HyperTalk Bible stack and name it `Fill`.
3. Enter the following handler in the button.

```
on mouseUp -- Fill button
  choose text tool
  set textFont to Geneva
  set textSize to 24
  set textStyle to bold
  set pattern to 22
  click at 100,75
  type "HyperTalk!!"
  choose select tool
  drag from 99,50 to 250,80 with optionKey
  doMenu "Fill"
  choose browse tool
end mouseUp
```

4. Click the Clear button and then click the Fill button.

Result:

The `drag with optionKey` version of the select tool selected just the black pixels in the drag rectangle. The `set textStyle to bold` produces bolded text, like that shown in Figure 17-15A. The `doMenu "Fill"` command produces text similar to that shown in Figure 17-15B (depending on your font selection).



A. Text in bold. B. Text after `doMenu "Fill"`.

Figure 17-15. Fancy text effects.

The Lasso Tool

The lasso tool has only one direct simulation use. Used with the `click at x,y with commandKey` command, lasso can select all the black pixels on the current

graphic layer. If the area clicked is completely surrounded by black pixels (a border, for instance), the outer boundary and the enclosed black pixels are selected. If you think of the black pixels as land, the white or transparent pixels surrounded by black as lakes, and the remaining white and transparent areas as ocean, the lasso tool can be used to select land and lakes but not ocean. Figure 17-16 shows a dramatization of the land, lakes, and ocean concept.

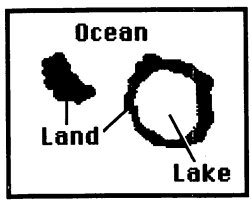


Figure 17-16. Pixel geography.

Figure 17-17A shows two separate objects, Figure 17-17B shows the circle selected after clicking inside it `with commandKey`, Figure 17-17C shows the objects pushed together to form one object, and Figure 17-17D shows the combined objects selected by clicking `with commandKey` inside the circle (note the cursor changes from the usual lasso to the pointer when over a selected area). In this example, the circle and the rectangle borders are land, the areas inside the borders are lakes, and the area outside is ocean.

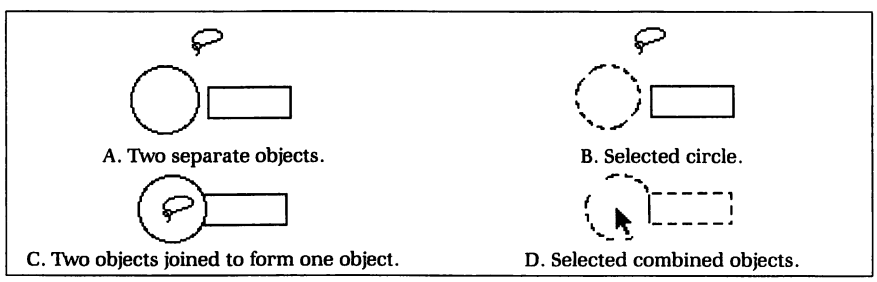


Figure 17-17. Lasso tool click results.

The doMenu "Select" *Command*

To lasso specific areas of the screen that are not completely bounded by black, you must use the select tool and `drag with optionKey`, as explained in the select tool description earlier in this chapter. Another way to lasso specific graphic elements is with the `doMenu "Select"` command. There are three ways to use this command. If your `doMenu "Select"` command immediately follows the creation of a graphic element with a paint tool, that element is lassoed. If you select an area with the select tool and then use the `doMenu "Select"` command, just the black pixels (and the lakes)

in the selection are lassoed. If nothing is selected and no graphics have been created since the last mouse click, `doMenu "Select"` lassos every graphic element in the active layer.

There are, unfortunately, two irregular cases with `Select`. One is that when paint text is selected, the lasso tool is used, but it doesn't shrink to fit the black pixels. In fact, the selected rectangle doesn't even tighten to the minimum rectangle size. If you want to lasso just the black pixels in graphic text, you have to drag the `select tool with optionKey`.

This irregularity is mostly a nuisance, but it has one silver lining when combined with the `doMenu "Invert"` menu command. Normally, if you use the `doMenu "Invert"` command with a lasso selection, the black pixels turn to white and the selection disappears. (Because no white pixels are lassoed, none are turned black). One exception is when you use the `click at... with commandKey` version of the lasso tool to select an enclosed area (a lake). In this case, `"Invert"` turns the enclosed area of white or transparent pixels black and the area's black border disappears.

Another exception is with text selected with the `doMenu "Select"` command. If you `doMenu "Invert"` this sort-of-lasso-selected text (remember, it doesn't shrink to fit), you get a very nice white-text-on-black-background effect.

Once an area is lassoed, it can be manipulated in a number of ways. Drag a selected region to reposition it. `drag with shiftKey` constrains the repositioning to vertical or horizontal. `drag with commandKey` resizes the selection. Dragging `with optionKey` clones a copy. Also, of course, the selection can be copied, cut, pasted, or cleared with the same menu commands listed in the previous section. Even the Rotate and Flip menu commands can be applied to the selection (a HyperCard improvement on MacPaint).

Example 17:

1. Open the Graphics card in your HyperTalk Bible stack.
2. Add a new card button to your HyperTalk Bible stack and name it `Lasso`.
3. Enter the following handler in the button.

```
on mouseUp -- Lasso button
  set centered to true
  set filled to false
  choose regular polygon tool
  set polySides to 3
  drag from 100,100 to 125,75
  set polySides to 8
  drag from 200,100 to 225,155
  set polySides to 5
  drag from 100,200 to 125,250
  set dragSpeed to 175
  choose lasso tool
  click at 100,100  with commandKey
  drag from 100,100 to 350,136
  click at 100,200 with commandKey
  drag from 100,200 to 300,250
  click at 200,100 with commandKey
```

continued

```
    drag from 200,100 to 50,250
    click at 350,136  with commandKey
    drag from 350,136 to 200,100
    click at 300,250 with commandKey
    drag from 300,250 to 100,100
    click at 50,250 with commandKey
    drag from 50,250 to 100,225
    choose browse tool
end mouseUp
```

4. Click the Clear button and then click the Lasso button.

Result:

First three polygons are drawn, then they are dragged individually back and forth across the screen.

Trouble with Dragging and Lasso

It is possible to get into trouble when dragging lasso-selected graphics. Because only the black pixels and enclosed areas of white or transparent pixels are actually selected, it is possible to have a starting drag coordinate that is in the middle of a selection without being part of the selection. To reposition a lasso-selected graphic from within a script with any confidence, you must either know the selection to be dragged well or have an XFCN that tells you whether a particular pixel is in a QuickDraw region (the technical name for a lasso selection). The problem here is that there may be unselected areas inside a complex region (the estuaries of the ocean alluded to before). If the starting drag coordinate is ocean rather than land or lake, instead of repositioning the selection, you either deselect it or make a larger selection (if you are dragging with commandKey).

Problems with *select* and Transparency

If you use the lasso tool to select every black pixel on the card layer (with `doMenu "Select All"` followed by `doMenu "Select"`, for instance), it also turns transparent every white pixel outside the selection (the ocean again). (This is perhaps the only instance in which merely selecting something permanently changes its appearance.) If you then use the `doMenu "Transparent"` command even the lakes will be turned transparent, and of course, you can use the `doMenu "Clear Picture"` command to make the entire graphic layer transparent.

The Eraser Tool

The eraser tool is a 16-pixel by 16-pixel square that turns everything it touches to white on the background layer and to transparent on the card layer. The eraser can only be used in one graphic layer at a time. It removes black pixels in either layer and removes white pixels in the card layer, allowing any background graphic to show through. Whether or not it actually erases white pixels on the background is a

pointless question, because the background itself is white. Dragging the eraser `with shiftKey` down constrains movement to horizontal or vertical.

Simulation Odds and Ends

There are a few menu commands that we haven't mentioned yet. Some of these commands were covered in the last chapter, but we say a little bit here about their use in graphic simulation.

The `doMenu "Undo"` command returns the screen to the way it was at the previous mouse click. There is one exception to this. The effects of a `doMenu "Clear Picture"` command are "ununDoable."

The `doMenu "Background"` command toggles between the card and background layers. It is better to use the `set editBkgnd` command for this purpose as it ensures that you get the graphic layer you want. With `doMenu "Background"`, there is a chance that you will toggle to the wrong layer if, for instance, one handler goes to the background and then calls another handler that also wants to go to the background.

The `doMenu "Import Paint"` command shows a list-type dialog box that lets the user pick an existing MacPaint file. Opening a file causes the image—or at least the top-left screenful of the file (if it was created in an application with a larger working surface than HyperCard's window)—to be pasted over the current graphic layer, thereby replacing any graphics that were there previously.

The `doMenu "Export Paint"` command stores a copy of the screen (minus the Menu bar and any shown windoids, but including buttons and fields) to a MacPaint file that the user must name.

HyperCard saves graphic changes to disk when you use the `doMenu "Keep"` command, change cards, move between graphic layers, or choose a general tool.

To dump recent graphic changes and return to the way a graphic layer looked the last time it was saved to disk, use the `doMenu "Revert"` command.

Menu Commands to Show Dialog Boxes

There are a number of menu commands that show dialog boxes used for setting painting properties. In most cases, you will set the properties directly in your scripts, but if you want to give users a chance to set things to please themselves, you will want to show some of these dialogs.

The `doMenu "Text Style"` command shows the Text Style dialog box and lets users change the text properties for graphic text. The font, size, style, alignment, and line height properties can all be set by clicking in this dialog box.

To show the user the Line Size dialog box, use the command `doMenu "Line Size"`. The line size is used by the line tool and all the shape tools. The box is hidden automatically when users set the size.

To let users choose a new brush shape, use the command `doMenu "Brush Shape"`. The dialog box (see Figure 17-18) is put away automatically when users pick a shape.

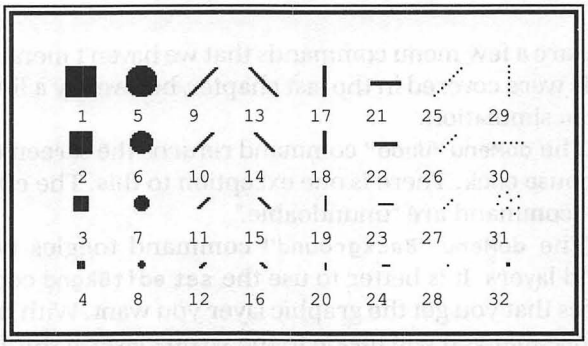


Figure 17-18. The Brush Shape dialog box.

Use the command `doMenu "Edit Pattern"` to let users edit the current pattern. Clicking the screen with the Edit Pattern dialog box showing causes the current pattern to be replaced with the pattern in the area of the click.

To let users change the polySides setting, use the command `doMenu "Polygon Sides"`. The dialog box is put away automatically when users pick a shape.

There are two final dialog box/windoids that you can show to users, but you get at the dialog boxes a different way. The Painting Palette (see Figure 17-19) and the Tool Palette can be shown with the commands `show tool window` and `show pattern window`. The windows can be hidden with the `hide` command. You can even move these windows around the screen by setting their `loc` properties, as in `set loc of tool window to 100,100`.

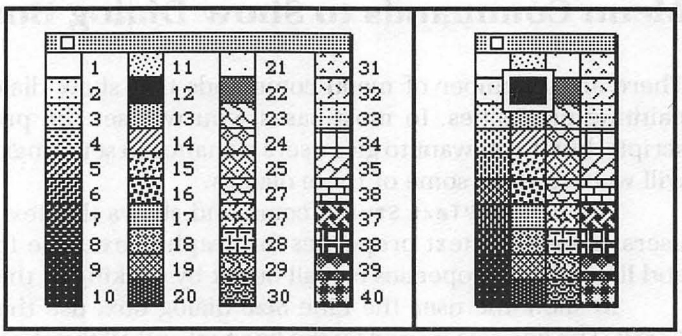


Figure 17-19. The Pattern Palette.

Showing and Hiding Pictures

One of the nicest features introduced with HyperCard version 1.2 is the capability of showing and hiding pictures independently of the card they are on. This means that for any card, you have four possible views: without any graphics, with background graphics only, with card graphics only, or with both card and background graphics.

Every background has a new `showPict` property that is true if the picture is visible and is false if the picture is not visible. Every card also has a `showPict` property that works the same way. You can use the `set` command to change the value of the property, or you can use one of two new commands. If you use the `set` command, you must specify which card or background you are working on. In most cases, you can say `set showPict of this card to true` or `set showPict of this background to false`.

The `show picture` and `hide picture` commands perform the same task as do the statements `set showPicture to true` and `set showPicture to false`. Like the other `hide` and `show` commands, the new `hide picture` and `show picture` commands make it a little easier to do what you could also do with the `set` command. The syntax for these new commands is shown in Figure 17-20.

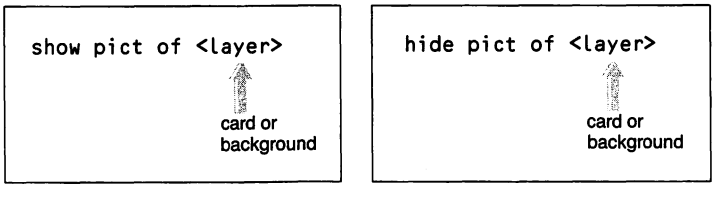


Figure 17-20. The `show pict` and `hide pict` commands syntax.

Example 18:

1. Open the Graphics card in your HyperTalk Bible stack.
2. Use the text tool (with the largest textSize you can manage) to paint the words `Card Graphic` on the card graphic layer and the words `Background Graphic` on the background graphic layer.
3. Enter the following statement in the Message box:

```
set showPict of this card to false
```

Result:

The words `Card Graphic` vanish from the screen.

4. Enter the following statement in the Message box:

```
show pict of card
```

Result:

The words `Card Graphic` reappear on the screen.

5. Enter the following statement in the Message box:

```
hide pict of bkgnd
```

Result:

The words `Background Graphic` vanish from the screen.

6. Enter the following statement in the Message box:

```
set showPict of this bkgnd to true
```

Result:

The words `Background Graphic` reappear on the screen.

In the next example, we show you a sample of the old-style card flipping animation.

Example 19:

1. Open the Graphics card in your HyperTalk Bible stack.
2. Add a new background to the stack. Copy the graphics from Figure 17-21 with the image in Figure 17-21A (the nucleus) going into the background graphic layer and Figure 17-21B (the electrons in phase 1) going into the card layer of the first card. Add a second card to the background and copy Figure 17-21C (the electrons in phase 2) into its card graphic layer.

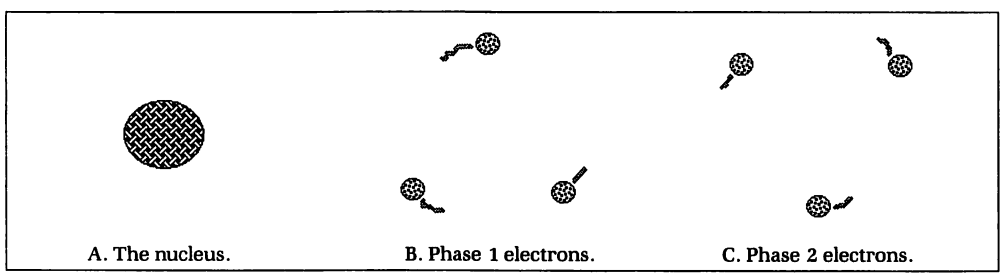


Figure 17-21. Graphics for Example 19.

3. Add a new card button to the first card of this new background and name it `Old Animation`.
4. Enter the following handler in the button.

```
on mouseStillDown -- Old Animation button
  go next
  go prev
end mouseStillDown
```

5. Press the mouse button in the Old Animation button and keep it pressed.

Result:

As long as the mouse button is pressed, the screen view flips back and forth between the two cards. The electrons appear to circle the nucleus.

Most HyperCard animation is like Example 19, in that it involves flipping through cards with one image per card. The next example shows how you can get multiple images from a single card with the `showPict` property.

Example 20:

1. Open your HyperTalk Bible stack and go to the new background you made in the last example.
2. Add a new background to the stack. Copy the graphics from Figure 17-22 with the image in Figure 17-22A (phase 1) going into the background graphic layer and Figure 17-22B (phase 2) going into the card layer.

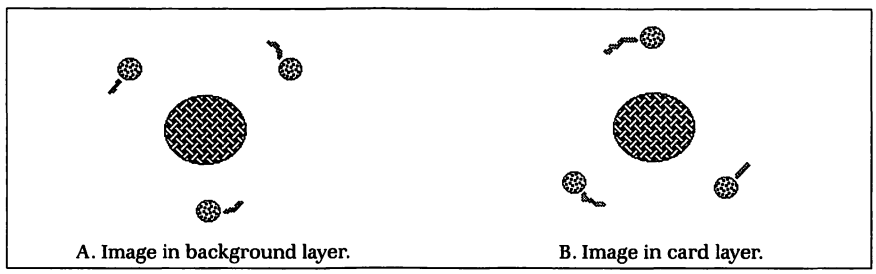


Figure 17-22. Graphics for Example 20.

3. Add a new card button to the first card of this new background and name it `New Animation`.
4. Enter the following handlers in the button.

```
on mouseDown -- New Animation button
  show pict of card
end mouseDown

on mouseStillDown
  lock screen
  set showPict of this bkgnd to not (the showPict of this bkgnd)
  set showPict of this card to not (the showPict of this card)
  unlock screen
  lock screen
  set showPict of this card to not (the showPict of this card)
  set showPict of this bkgnd to not (the showPict of this bkgnd)
  unlock screen
end mouseStillDown

on mouseUp
  hide pict of card
end mouseUp
```

5. Press the mouse button in the New Animation button and keep it pressed.

Result:

As long as the mouse button is pressed, the screen view flips back and forth between the two cards. The electrons appear to circle the nucleus.

Of course, the `showPict` property is not only for animation. You can now use graphic text in place of pop-up fields on cards that don't use other graphics. You can put special instructions or credits into the graphic layer and only show them when the user clicks a button. Because there are two separate graphic layers, you can even

toggle between two information screens using them like keyboard templates. The possibilities are endless.

Pasting into Hidden Pictures Is a Bummer
If you attempt to paste a picture into a hidden graphic layer, a dialog box appears, asking whether you want to show the picture. To get the picture into the graphic layer, you must click OK in the dialog box and then repeat the paste operation. When you are writing scripts that paste pictures, you might consider adding a line to set the `showPict` to true prior to a `doMenu "Paste Picture"` command. This could save the user some confusion.

Summary Script

Now that you have seen all the Graphic simulation tools, we can try a script that combines lots of tools and properties.

In this example, we store a list of options in variables and then choose from the list at random. We rename the button with a message telling the user how to stop the action. (You can't click the button because the browse tool is not chosen.) The drag coordinates are picked at random and may result in graphics that extend off the window. This example creates totally random graphics and maybe even art.

Example 21:

1. Open the Graphics card in your HyperTalk Bible stack.
2. Add another wide, new card button to your HyperTalk Bible stack and name it `Random Art`.
3. Enter the following handler in the button.

```
on mouseUp -- Random Art button
  put "pencil tool,brush tool,eraser tool,line tool," into toolVar
  put "spray tool,rectangle tool,round rect tool,oval tool,"¬
   after toolVar
  put "brush tool,regular polygon tool,regular polygon tool"¬
   after toolVar
  set dragSpeed to 200
  set the name of card button the short target to "Stop With Option Key"
  put 1 into n
  show tool window
  show pattern window
  repeat until the optionKey is down
    choose item random(11) of toolVar
    if the tool is "regular polygon tool" then
     set polySides to random(6) + 2
    end if
    set lineSize to random(4)
    set pattern to random(39) + 1
    put random(400) + 50 into x
```

continued

```
    put random(250)  + 50 into y
    if n mod 2 = 0 then
      drag from x ,y to x + random(100),y + random(100)
    else
      drag from x ,y to x + random(100),y - random(100)
    end if
    if the tool is "brush tool" then
      repeat 30
        if the optionKey is down then
          choose browse tool
          set the name of card button the short target to "Random Art"
          exit mouseUp
        end if
        set brush to random(32)
        set pattern to random(39) + 1
        put random(510) into x
        put random(340) into y
        click at x,y
      end repeat
      set pattern to 12
    end if
    set centered to not the centered
    set filled to not the filled
    put n + 1 into n
  end repeat
  choose browse tool
  set the name of card button the short target to "Random Art"
end mouseUp
```

4. Click the Clear button and then click the Random Art button.

Result:

Depending on how long you run this handler, you should get a random pattern of various shapes, brush shapes, and patterns (see Figure 17-23).

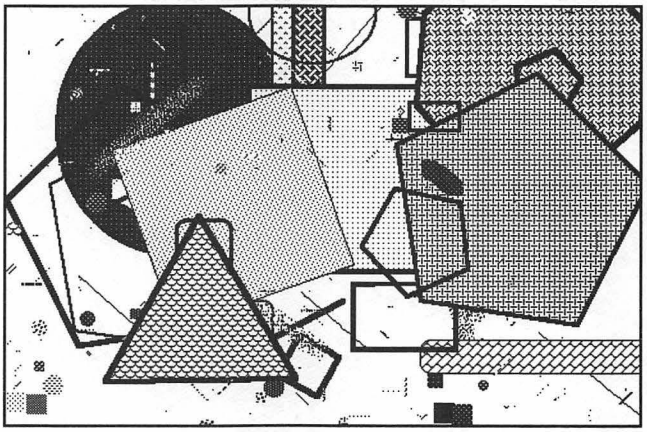


Figure 17-23. Example 21 result after the better part of a minute.

You don't have to show the tool and pattern windows, but if you do show them (and also watch the cursor changes), you get a better idea of what is happening. Example 21 uses most of the techniques from Example 20 but adds tool changing and an inner repeat 30 loop for scattering brush dabs on the screen.

Review Questions

1. What determines the color of a line drawn with the pencil tool?
2. How would you draw a patterned line with the line tool?
3. How would you use the brush tool as an eraser?
4. What command has to precede a `type` command when you are typing paint text to the screen with the text tool?
5. List the properties that effect the appearance of rectangles drawn with the rectangle tool.
6. Which shape tool always draws centered?
7. How do you lasso select all the graphics as a specified rectangular area of a graphic layer?
8. How do you lasso select a shape, line, dot, or graphic text string just after it has been created?
9. How do you create inverted text (white letters on black)?
10. How do you turn all the white pixels in a selection transparent?
11. Are there any HyperCard dialog boxes you can't show from a script?

Exercises

1. Write a script that completely blacks out the screen in three passes using one-pixel-wide horizontal lines.
2. Do a second version of the above script to black out the screen in two passes with three-pixel-wide horizontal lines.

Chapter 18

Global Properties

Quick View

Commands
lock screen
unlock screen

Properties
blindTyping
cursor
dragSpeed
editBkgnd
language
lockScreen
powerKeys
userLevel
lockRecent
lockMessages
userModify
location, loc
rectangle, rect
visible
left
right
top
bottom
topLeft

Properties *cont'd*
bottomRight, botRIght
textarrows

Keywords
tool window
pattern window
card window

Functions
the screenRect

Concepts
Global properties
User preferences card and userLevel
Using the textArrows property
Shortcuts with power keys
Using cursor properties
Using lockScreen, lockRecent, and lockMessages
Write protecting stacks
Window properties
Using rectangle and screenRect properties
Fancy use of rectangle property

Chapter 18

Global Properties

Do you want to set up a stack that automatically places users on the painting level and opens the tool and pattern windows when users open the stack? Do you want to freeze the screen while a script works behind the scenes, have the Message box automatically displayed when a particular card is opened, or set up a tutorial stack that users can alter temporarily but not permanently? You can do all these things and more by using properties.

Properties are one of HyperTalk's central ideas. In general, a property is a changeable characteristic of an object, for instance, the name of a card or the location of a button. Most properties can be obtained by using the `get` command and reset using the `set` command. We can classify properties by the objects they apply to. For example, there are stack properties and button properties. You already encountered most types of properties earlier in this book; in this chapter, we look at global properties (which apply to all your stacks) and window properties (which apply to the card window, the tool window, the pattern window, and the message window).

Global Properties

Global properties apply to the whole HyperCard environment. This means if you set a particular global property, the setting holds for all your stacks and their contents. For instance, your User Preferences card (found at the end of the Home Stack) actually sets global properties to describe your user level along with some other qualities. When you select a user level, you keep that level as you change from stack to stack, unless you or a script reset that level. Because global properties hold for the whole HyperCard system, they also are called system properties.

Global properties work like the other properties you've seen. If you want to find out the value of a global property, in most cases, you can simply type its name in the Message box. For example, if you type `the userLevel` in a Message box, the Message box then displays the current setting. You can also type `get the userLevel`, and HyperTalk places the value into the global variable `it`. To reset a global property, use the `set` command, as in `set the userLevel to 3`.

Let's look at the global properties now.

User-Preference Properties

Most likely one of the first things you learned about HyperCard is going to the User Preferences card to select your preferences. Normally, that card (Figure 18-1) is the last card in the Home stack. In it, you select settings in four categories: User Level, Text Arrows, Power Keys, and Blind Typing. (The Text Arrows setting was added to the card in HyperCard version 1.1.)

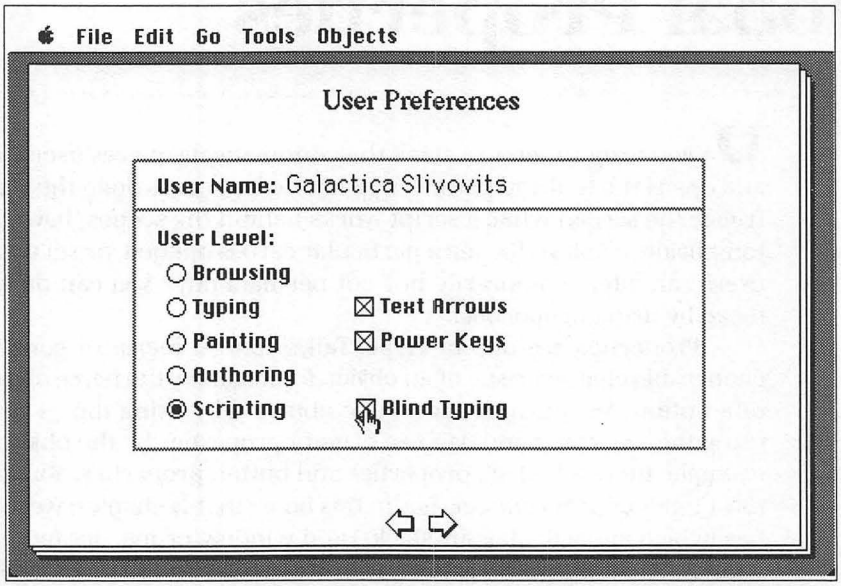


Figure 18-1. The User Preferences card.

HyperTalk lets you access and alter these four settings through four global properties: `userLevel`, `textArrows`, `powerKeys`, and `blindTyping`. By using the `get` command with these properties, you can obtain the current settings. By using the `set` command, you can change them. Let's examine them in turn.

The userLevel *Property*

It's pretty obvious that this property describes the user level, which controls the degree to which you can use and alter a stack. Let's see it in action.

Example 1:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Enter

```
the userLevel
```

Result:

The Message box displays 5. (This assumes that you have selected the scripting level; if you haven't, you may have had some problems getting this far into the book.)

Note that the `userLevel` property describes the level by number rather than by name. HyperTalk uses the correspondences shown in Table 18-1.

Table 18-1. `userLevel` **Property Settings**

Level Name	Level Number	Meaning
Browsing	1	Can move through stack and read text
Typing	2	Can also enter text
Painting	3	Can also use the paint tools
Authoring	4	Can also use the button and field tools
Scripting	5	Can write and modify scripts

You can use the `userLevel` property, for example, to turn off the scripting level for a particular stack or card.

Example 2:

1. Go to your HyperTalk Bible stack and create a new card called `Simple Card`.
2. Place the following text into Simple Card's script:

```
on openCard
  global level
  put the userLevel into level
  set the userLevel to 1
end openCard

on closeCard
  global level
  if level <> empty then set the userLevel to level
end closeCard
```

3. Leave the card and return to it.

Result:

The Menu bar only displays the File, Edit, and Go choices, as shown in Figure 18-2.

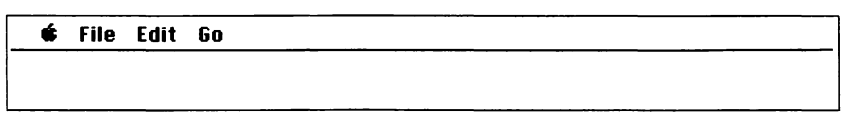


Figure 18-2. Menu bar after setting `userLevel` to 1

4. Leave the card again.

Result:

The Menu bar is restored to its full list of selections.

When you first leave the card, the global variable `level` hasn't been set yet, because you entered the openCard handler after the card had been opened. Therefore, the closeCard handler is instructed to do nothing if `level` is empty. When you return to the card, the openCard handler is run, the global variable `level` stores the current level, and the user level is set to browsing. This time, when you leave the card, `level` holds the original user level, and, in the closeCard handler, the level is reset to that value.

What if you want to modify the script for the Simple card? Because the user level is 1, the Objects heading is missing from the Menu bar, and you can't pull down the Card Info... selection. The solution is to open the Message box and enter `set the userLevel to 5`. Then, you are back in the scripting business again.

The textArrows *Property*

Starting with HyperCard Version 1.1, a new item was added to the User Preferences card: Text Arrows. If you check this setting, you can use the arrow keys on your keyboard to move through the text. Press an arrow key, and the cursor moves in the direction the arrow indicates. If the setting is not checked, the arrow keys change which card you are viewing. (The right arrow moves you to the next card in the stack, the left arrow moves you to the previous card in the stack, the up arrow moves you back through the cards you've viewed in reverse order, and the down arrow brings you forward through the cards you've viewed.) If you select the Text Arrows choice, you still can use the arrow keys to move through the stack by using the option key. Thus, when Text Arrows is not selected, the right arrow key takes you to the next card in the stack. If Text Arrows is selected, use Option-Right arrow for the same effect.

The `textArrows` property is true if you select Text Arrows and false otherwise. You can set this property on the User Preferences card, and you can set it from the Message box or a script.

Example 3:

1. Go to the User Preferences card of your Home stack and select Text Arrows.
2. Open the Message box and enter

```
the textArrows
```

Result:

The Message box displays `true`.

To change the setting, you can use either the Message box or the `set` command.

3. Enter in the Message box

```
set the textArrows to false
```

4. Leave the User Preferences card (use the left arrow key if you have one), and then return to it.

Result:

The Text Arrows box no longer is checked.

The change took effect as soon as you gave the `set` command, but the User Preferences card wasn't updated until you forced it to be redrawn.

Suppose that you create a stack and you want the arrow keys to be used for navigation only. For example, you could want an informational stack intended for browsing rather than for text entry. Put the following handlers into the stack script:

```
on openStack
  global arrows
  put the textArrows into arrows
  set the textArrows to false
end openStack

on closeStack
  global arrows
  if arrows <> empty then set the textArrows to arrows
end closeStack
```

This is analogous to Example 2. Opening the stack sets the `textArrows` property to the value you want, and closing the stack restores the value to the user's setting.

The powerKeys *Property*

If you select Power Keys on the User Preferences card, you activate many one-key shortcuts that you can use when in Painting mode. Table 18-2 provides a list of these shortcuts. Pressing the C key, for example, toggles centered drawing on and off. You should have the Message box hidden when you use the power keys; otherwise the keystrokes are interpreted as Message box input. That is, if you press the C key with the Message box open, you get a `c` in the Message box. However, there are two keystroke commands you can use whether the Message box is closed or open: Command-A (Select All) and Command-S (Select). So pressing the A key invokes the Select All command only if the Message box is closed, but Command-A invokes the same command regardless of the Message box state.

When you select the text tool, the power keys are inactivated. This lets you enter text on the paint level.

The blindTyping *Property*

If the `blindTyping` property is true, you can type Message box commands even when the Message box is not showing. This is useful if you are a sure-fingered typist and want to keep the whole card visible. As with the preceding two properties, `blindTyping` is either true or false. You can type the property name preceded by `the`

to find its value and use either the User Preferences card or the `set` command to set the property.

Table 18-2. Power Key Shortcuts for Painting

Paint Menu Selections

Key	Command	Key	Command
A	Select All	P	Pickup
D	Darken	R	Revert
E	Trace Edges	S	Select
F	Fill	T	Transparent
H	Flip Horizontal	V	Flip Vertical
I	Invert	[	Rotate Left
L	Lighten	]	Rotate Right

Options Menu Selections

Key	Command	Key	Command
C	Draw Centered (on/off)	M	Draw Multiple (on/off)
G	Grid (on/off)	1,2,3,4,6,8	Line Size in pixels

Patterns Menu Selections

Key	Command	Key	Command
B	Black Pattern	W	White Pattern

Example 4:

1. Go to the User Preferences card in your Home stack and select Blind Typing.
2. Open the Message box and enter

```
the blindTyping
```

Result:

The box displays `true`.

3. Click the disappear box in the Message box and then (carefully) type the date.

Result:

The Message box disappears after you click the little box. After you type the date, the Message box reappears to display the date.

One obvious disadvantage of blind typing is that you can't see what you are typing, so poor typists should avoid this option.

4. Enter

```
set the blindTyping to false
```

5. Attempt to repeat Step 3.

Result:

The Macintosh beeps at you when you try to type.

If you have `powerKeys` set to true, `blindTyping` is disabled when you are in Painting mode.

The *cursor* Property (Version 1.1 and earlier)

The `cursor` property describes the appearance of the cursor. Unlike the other global properties, the `cursor` property can't be read. If you try entering `get the cursor` in the Message box, you are told that HyperTalk can't get that property. However, you can set `cursor`. HyperTalk provides several built-in choices, as shown in Figure 18-3.

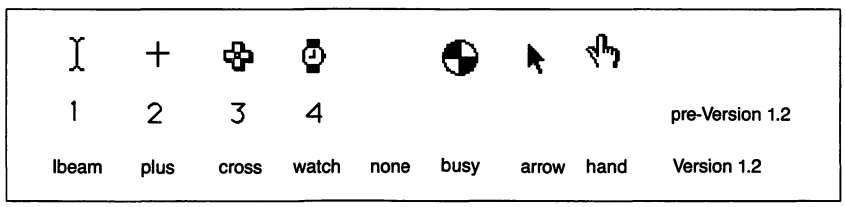


Figure 18-3. HyperTalk cursors.

For example, to get cursor #4, give the command `set the cursor to 4`. There is one catch to setting the cursor: the `idle` message resets the cursor to the original cursor. The `idle` message, recall, is sent repeatedly as long as the mouse is up and while no scripts are being run. To see the effect of this command, you have two choices: One is to type the command into the Message box, press the return key, and then quickly hold down the mouse button before the cursor can revert. This can be difficult to do. The second is to use the command in a script.

Example 5:

1. Go to your HyperTalk Bible stack and create a new card called `Global`.
2. Create a new button called `Cursor 1` and give the new button this script:

```
on mouseUp -- Cursor 1 button
  ask "Which cursor number? <1-4>"
  if it < 1 or it > 4 then exit mouseUp
  set the cursor to it
  wait until the optionKey is down
end mouseUp
```

3. Click the Cursor 1 button; a dialog box asks `Which cursor number? <1-4>`. Respond to the dialog box with `3`.

Result:

The cursor becomes the fat cross until you press the option key.

Pressing the option key terminates the script. Once the script finishes, the `idle` messages start again, resetting the cursor.

The *cursor* Property

Version 1.2 added four more cursor forms, along with names for all eight cursors. You can use these names rather than numbers when setting the cursor. Indeed, you must use the name for the four new cursors because they are not numbered. However, to maintain compatibility with earlier HyperTalk versions, you also can refer to the first four by number. Thus, with Version 1.2, you can use either `set the cursor to watch` or else `set the cursor to 4` to get the watch cursor, but the only way to get the hand cursor is to use `set the cursor to hand`.

The differences between versions of HyperCard pose some difficulties for scripts intended to work for all versions. Because HyperCard versions are upwardly compatible, Example 5 works on all versions, but it doesn't make the new cursors available. The next example also works on all versions, and it makes the new cursors available if you have the newer versions of HyperCard. The key is using the function `the version`. It returns the current HyperCard version number, allowing a script to match its behavior to the version.

Example 6:

1. Go to the Global card of your HyperTalk Bible stack and create a copy of the Cursor 1 button.
2. Rename the copy `Cursor 1.2` and give it this script:

```
on mouseUp -- Cursor 1.2
  put "1,2,3,4,none,busy,arrow,hand" into choices
  if the version < 1.2 then put 4 into limit
  else put 8 into limit
  ask "Which cursor number? <1-" & limit & ">"
  if it < 1 or it > limit then exit mouseUp
  set the cursor to item it of choices
  wait until the optionKey is down
end mouseUp
```

3. Click the Cursor 1.2 button and respond to the dialog box with `8`.

Result:

The cursor becomes the hand until you press the option key.

As promised, this script works with earlier versions of HyperTalk as well as with Version 1.2. If the version is earlier than 1.2, the script limits the cursor choices to the first four. We used the numbers rather than the names for these first four cursors because all versions recognized the numbers, whereas the earlier HyperCard versions don't recognize the names. One other change we made is to place a list of cursor names into the variable `choices`. Then, we use the value of the global variable `it`, which holds the user input, to identify which item in the list to use.

Removing a Button's Icon
If you have a button set to a particular icon and want to change it manually, go to the button's dialog box, press Icon, and then press None when the icon list is displayed. You can accomplish the same thing from a script by using the statement

```
set icon of cd btn 1 to zero
```

The busy cursor differs from the rest by coming in different "orientations." You can't choose which one, but by setting the cursor repeatedly, you select each orientation in turn, making the cursor appear to rotate.

Example 7:

1. Go to the Global card of your HyperTalk Bible stack.
2. Create a button called `Busy` and give it this script:

```
on mouseUp -- Busy button
  repeat until the optionKey is down
    set the cursor to busy
  end repeat
end mouseUp
```

3. Click the Busy button.

Result:
The beach ball cursor appears to spin until you hold down the option key.

What's actually happening is that you are seeing the beach ball in four different orientations shown in succession. This kinetic effect is more eye-catching than the simple watch, and it better conveys the impression that the computer is working away at some task.

The *dragSpeed* Property

When you use the `drag` command in scripts to paint, HyperTalk normally moves whatever is being dragged as fast as it can. The `dragSpeed` property, however, can be used to control the drag rate. The drag rate is calibrated in pixels per second, so a command like `set the dragSpeed to 100` would let a script move, say, the pencil tool at 100 pixels per second. Setting `the dragspeed` to 512 would let you cross the screen width in one second.

Why would you want to drag at less than top speed? One reason would be to make a scripted drawing appear on a card slowly enough to produce the effect of an invisible hand drawing. Alternatively, you may want to drag images across the screen to produce an animation effect. For example, the Clip Art stack includes a picture of an old automobile that you can make move by clicking a button. If you look at the script, you see it sets a `dragspeed` of 100 to produce a properly sedate motion.

The default value for dragSpeed is 0. This may sound rather slow, but HyperTalk interprets 0 to mean "as fast as I can." This limit depends on your hardware; a Macintosh II outdraws a Macintosh Plus. Here's a short demo of what different speed settings produce.

Example 8:

1. Go to the Global card of your HyperTalk Bible stack and create a button called Draw Lines.
2. Put the following script into the button:

```
on mouseUp -- Draw Lines button
  choose pencil tool
  repeat with m = 1 to 2 -- trace the figure twice
    repeat with n = 0 to 4 -- 5 horizontal lines
      set the dragSpeed to n * 50 * m * m
      put the dragspeed into the message box
      drag from 128,120 + n * 25 to 384, 120 + n * 25
    end repeat
    repeat with n = 0 to 8 -- nine vertical lines
      set the dragspeed to n * 50 * m * m
      put the dragspeed into the message box
      drag from 128 + 32 * n,121 to 128 + 32 * n,219
    end repeat
  end repeat
  choose browse tool
end mouseUp
```

3. Click the Draw Lines button.

Result:

The Macintosh draws a pattern (see Figure 18-4) at varying rates of speed, and then erases it line by line, also at varying rates of speed. The Message box displays the current speed.

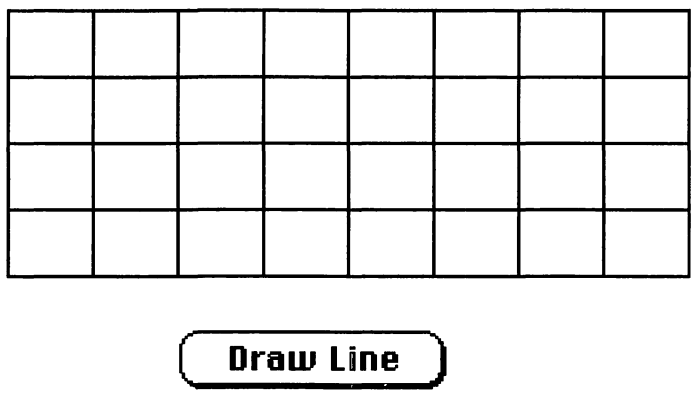


Figure 18-4. Using dragSpeed.

The first pass of the outer loop (`m = 1`) draws the figure. The second pass (`m = 2`) erases it, because if you start the pencil tool on a black pixel, it draws in white.

Note that the first horizontal and the first vertical lines are drawn very fast. That's because the first time through each loop, `n` has the value 0, so the drag speed (`n*50*m*m`) is 0. Then, the drag speed is set to 50, 100, 150, and so on while the figure is being drawn. This is the first pass through the outer loop, so `m` and `m*m` are 1. The second pass has `m = 2`, so `m*m` is 4. This causes the drag speeds to be set to 200, 400, 600, and so on, so the erasure goes four times faster than the drawing.

The *editBkgnd* Property

When you are painting or creating fields, buttons, or the like, you can switch between the card level and the background by using the Background toggle in the Edit menu or by using the Command-B keystroke combination. You also can use the `editBkgnd` property, either from the Message box or from a script. This property is set to true if you are in the background and false otherwise.

Example 9:

1. Go to your HyperTalk Bible stack; stay at the card level.
2. Enter `editBkgnd` in the Message box.

Result:

The Message box displays `false`.

3. Enter

```
set the editBkgnd to true
```

in the Message box.

Result:

You are placed in the background level.

While you are in the background level, create a background button that toggles you back and forth between the card and background. (If you place the button on the card level, it disappears when you go to the background level, but a background button is visible on both levels.)

Example 10:

1. Create a background button for the Global card and give it the name `Bkgnd to Off`.
2. Give the button this script:

```
on mouseUp -- Bkgnd to Off button
  if the editBkgnd is false then
    set the editBkgnd to true
    set the name of me to "Bkgnd To Off"
  else
```

continued

```
    set the editBkgnd to false
    set the name of me to "Bkgnd To On"
  end if
end mouseUp
```

3. Click the button a few times.

Result:

Clicking the button toggles you between the card and the background and changes the button name.

If you're working on a project that involves switching back and forth frequently between card and background, this button could save you a little effort, as it is marginally more convenient than the alternatives.

We look at another example of using the `editBackground` property when we cover the `lockScreen` property.

The *language* Property

HyperCard comes in several different language versions. These are human languages (English, French, etc.), not computer languages. The `language` property reveals and lets you select which language is being used. Most users only have one choice, but some have more.

The `language` property is set to the name of the language.

Example 11:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Enter

```
the language
```

Result:

The Message box displays `English`, or, possibly, another name.

3. Enter

```
set the language to Romanian
```

Result:

An answer box states `Don't have a translator for that language.`

If you are writing stacks for international distribution, you can use this property to match the stack to the language being used.

The *lockScreen* Property

To lock the screen means to freeze the screen appearance while other HyperTalk processes go on. For example, you may have a program that goes to a field on an-

other card, retrieves some information, and then returns to the original card. If the screen is unlocked, you see these activities take place before your eyes. If it is locked, the activities are hidden from view. This spares the user from watching irrelevant screen events and speeds up the processing. The `lockScreen` property controls HyperCard's behavior in these situations. If the `lockScreen` property is set to true, the screen is locked, or frozen. If `lockScreen` is set to false, the screen can be redrawn.

You could get into lots of trouble if you could keep the screen locked. For that reason, the `lockScreen` property automatically is reset to false on the `idle` message. This message is not sent while a script is running, so it won't interfere with using `lockScreen` in a script. If you neglect to set `lockScreen` to false before a script finishes, however, an `idle` message does it for you soon after the script ends.

There's not much point to experimenting in Message box with this command, so let's go directly to a script example. We start with a script that doesn't lock the screen.

Example 12:

1. Go to the Global card of your HyperTalk Bible stack.
2. Create a button called `Dog Names` and give it this script:

```
on mouseUp -- Dog Names button
  push this card
  go to card "Pet Names"
  put random (the number of lines of card field "Dogs") into¬
  choice
  put line choice of card field "Dogs" into name
  pop card
  answer "Call your pet" && name
end mouseUp
```

3. Create a new card called `Pet Names`.
4. Create a card field called `Dogs` on the Pet Names card and give it the following text. (Use the return key so that there is one name per line.)

```
Lassie
Rin Tin Tin
Woobles
Frisky
Duke
```

5. Return to the Properties card and click the Dog Names button.

Result:

The Pet Names card appears on the screen briefly, and then the original card reappears as an answer box and shows you the chosen name.

Now, let's use `lockScreen` to modify the behavior of this script.

Example 13:

1. Go to the Global card of your HyperTalk Bible stack.

2. Modify the Dog Names button script to read as follows:

```
on mouseUp
  set the lockScreen to true -- freeze the screen
  push this card
  go to card "Pet Names"
  put random (the number of lines of card field "Dogs") into¬
   choice
  put line choice of card field "Dogs" into name
  pop card
  answer "Call your pet" && name
  set the lockScreen to false -- reactivate screen drawing
end mouseUp
```

3. Click the Dog Names button.

Result:

This time there is no card changing as the script does its work.

The `lockScreen` property is particularly handy when the script has to examine several cards or visit other stacks. Not only does it make the screen less distracting, it speeds the script because the Macintosh doesn't have to spend time updating the screen.

The *lock screen* and *unlock screen* Commands

Version 1.2 of HyperCard adds two new commands, `lock screen` and `unlock screen`, that provide an alternative method for controlling the screen. The `lock screen` command is equivalent to saying `set the lockScreen to true`, and the `unlock screen` command is the same as saying `set the lockScreen to false`. That is, the new commands' effect is to set the `lockScreen` property. The new commands are shorter than the equivalent `set` commands, but the `set` commands work with all versions of HyperCard.

The *lockRecent* Property

As you navigate through HyperCard's stacks and cards, HyperCard maintains a record of the cards you've visited most recently. When you select Recent from the Go menu, HyperCard displays a card showing miniatures of these cards. The `lockRecent` property controls whether or not visited cards are added to the Recent list. If the property is set to true, they are not added; if `lockRecent` is false, visited cards are added.

The default value for `lockRecent` is false, so normally, cards are added to the Recent list. You may not want this to be the case. For instance, you may have a stack in which you want card access controlled by a set of buttons, You may want the Recent card to reflect just the first cards of the stacks you visit and not all the interior cards visited, or a stack might search through several cards before finding the de-

sired card, and you don't want the irrelevant cards displayed on the Recent card. By using `lockRecent`, you gain some control of these situations. Setting `lockRecent` to true also speeds some processes because HyperCard doesn't have to take time to add cards to the recent list.

Like `lockScreen`, `lockRecent` is reset to false by the `idle` message, so it doesn't do much good to set this property to true in the Message box. Instead, `lockRecent` is intended for use in scripts. If you are setting `lockScreen` to true, don't worry about `lockRecent`, because cards visited while the screen is locked aren't added to the recent list in any case. A typical situation in which to use `lockRecent` is a script that shows the user several cards (an animation, for example) before returning to the original card. By using `lockRecent`, you don't clutter the user's Recent card with a lot of secondary information.

Here's a short example showing how the property works.

Example 14:

1. Go to the Global card of your HyperTalk Bible stack.
2. Create a button called `Quick Tour` and give it this script:

```
on mouseUp -- Quick Tour button
  set the lockRecent to true
  show all cards
  set the lockRecent to false
end mouseUp
```

3. Click the Quick Tour button.
4. Select Recent from the Go menu.

Result:

Clicking the button causes HyperTalk to show all the cards in your HyperTalk Bible stack, but they are not all added to the Recent card.

For a large stack, using this Quick Tour button keeps your Recent card much less cluttered than does using `show all cards` directly from the Message box. (If you like, try the latter approach; it places as many cards as fit into the Recent card, possibly dislodging previous entries.)

The *lockMessages* Property

HyperCard constantly is generating system messages: the `openStack` message, the `mouseDown` message, the `openCard` message, and so on. Setting the `lockMessages` property to true blocks these system messages, and setting it to false turns them back on. Setting the property to true is useful when you don't want these system processes to interfere with a script. Like the other global locking properties, this property is reset to false by the `idle` message.

To see how this property works, let's start by modifying Example 13.

Example 15:

1. Go to the Pet Names card of your HyperTalk Bible stack.

2. Place the following handler into the card script:

```
on openCard
   play "harpsichord" a c a
end openCard
```

3. Go to the Properties card and click the Dog Names button.

Result:

Besides showing you a name for your dog, your Macintosh plays a three-note sequence.

What's happened here is that even though the `lockScreen` property hides the Pet Names card from us, that card still is opened by the `go to card "Pet Names"` command in the button script. That generates the `openCard` message, which in turn, triggers the openCard handler in the card script.

Now, let's use `lockMessages` to suppress that message:

Example 16:

1. Go to the Properties card of your HyperTalk Bible stack.
2. Modify the Dog Names button script to read as follows:

```
on mouseUp
  set the lockScreen to true
  set the lockMessages to true -- new line
  push this card
  go to card "Pet Names"
  put random (the number of lines of card field "Dogs") into¬
   choice
  put line choice of card field "Dogs" into name
  pop card
  answer "Call your pet" && name
  set the lockMessages to false -- new line
  set the lockScreen to false
end mouseUp
```

3. Click the Dog Names button.

Result:

Your Macintosh suggests a dog name but doesn't play the harpsichord notes.

Because the `lockMessages` property was set to true, the `go to card "Pet Names"` command did not generate an `openCard` message. Thus, the on openCard handler in the Pet Names card script was not activated.

A handy use for `lockMessages` is editing the script for a stack that executes a bad script on the `openStack` message. Normally, you would open a stack to edit its script, but the act of opening such a problem stack sends a message that activates the bad script. Suppose, instead, that you use a script along these lines:

```
on mouseUp
  set lockMessages to true
  go to stack "Messed Up"
```

continued

```
  edit the script of this stack
end mouseUp
```

Because `lockMessages` is true, the `openStack` message is blocked, and you can edit the stack without activating the bad script.

You can use a similar approach if you need to edit a bad card script that is enacted on the `openCard` message.

Amazingly Useful Gesture Script

Here is one of the most interesting ideas for a HyperTalk script we've seen yet. You use it to open the script of a button without having to change to the button tool. To use it, put the script inside of the Home stack and move the cursor two times in and out of the top edge of the button. If you make this "gesture" within two seconds, the script editor opens with the button's script showing.

```
on mouseStillDown
  if "button" is not in the name of the target then exit
mouseStillDown
  global editTicks, editState
  if (the ticks) - editTicks > 120 then
    put the ticks into editTicks
    put 0 into editState
  else
    if the mouseV < top of the target then
    if editState = 0 then put 1 into editState
    if editState = 2 then put 3 into editState
  endif
  if the mouseV > top of the target then
    if editState = 1 then put 2 into editState
    if editState = 3 then put 4 into editState
  end if
  if editState = 4 then edit script of the name of the target
end mouseStill Down
```

Note that the user has to move the mouse up and down two times within 120 ticks (two seconds) for the variable `editState` to accumulate enough points (4) to cause the script to open. You can change the gesture "edge" by changing the statement `...mouseV < top to...` to `...mouseH < left of...` or whatever you wish. Ted Kaehler, one of the members of the HyperCard development team, came up with the idea for this nifty script.

The *userModify* Property

HyperCard provides for "write-protected" stacks. These are stacks you can read but not write in. Version 1.2 of HyperCard also supports "locked files," which are write-protected files that can be read by more than one user at one time. An example is a CD-ROM serving several Macintoshes. The important point here is that you can't alter locked or write-protected stacks or files unless you first remove those protec-

tions. With the help of the `userModify` property (introduced with Version 1.2), you can make temporary changes that are not saved. For example, you might prepare a tutorial stack that encourages the user to try the paint tools or to enter data in fields. However, you may want to make sure that stack ultimately is left in its original condition. To do this requires two steps. First, lock or write-protect the stack. Second, set the global property `userModify` to true. Let's see how to take these steps.

There are several ways to write-protect a stack. One is to select the Protect Stack... choice in the File menu. That opens a dialog box which contains a Can't Modify box that you can check. Second, Version 1.2 introduced a stack property called `the cantModify`. Setting this property to true has the same effect as checking the Can't Modify box. Third, if you are in the finder, you can select the stack and choose Info from the Finder's File menu. That opens a dialog box containing a Locked box you can check. The first two methods can be reversed from inside the stack, but you have to go to the finder to unlock a stack once it's been locked. Beginning with Version 1.2, write-protected HyperCard stacks show a small padlock to the right of the menu headings in the HyperCard menu bar.

Setting the `userModify` property is a matter of using the `set` command. Let's examine an example.

Example 17:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Enter

```
set the cantModify of this stack to true
```

3. Try adding text to one of the fields.

Result:

After you give the `set` command, a lock symbol appears on the Menu bar. When you try to add text, a dialog box states `Can't modify this stack`.

If you investigate further, you find you can't move buttons or alter scripts. If you open the Objects menu, you find you can't create new objects. Yes, the stack is write-protected.

Now let's try out the `cantModify` effect.

Example 18:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Enter

```
the cantModify of this stack
```

Result:

The Message box shows true. If not, abort this example.

3. Enter

```
set the userModify to true
```

4. Select the brush tool to true and paint a pattern on the card.

Result:

You paint a pattern on a card.

5. Go to another card and then return to the first card.

Result:

Your handiwork has disappeared.

The changes you make hold only as long as you stay in a card. Leaving a card restores it to its original condition.

6. Find a card with a non-locked text field and add some text.

Result:

You can add text to the field.

Also, you can move fields and buttons around and create new ones. However, you can't give scripts to new buttons and fields, you can't modify existing scripts, and you can't create new cards or backgrounds. In short, you are limited to adding and modifying text in old and new fields, moving objects around, and using the paint tools.

7. Go to another stack, and then return to the HyperTalk Bible stack.

Result:

The stack is in its original condition; your changes were not saved.

8. Enter in the Message box

```
the userModify
```

Result:

The Message box shows `false`.

The last result indicates that the `userModify` property is reset to false when you switch stacks. It's also reset to false when you quit HyperCard. To make the property hold each time you enter the stack, place the `set userModify to true` command into an openStack handler in the stack script.

Once you finish this example, enter `set cantModify of this stack to false` in the Message box so that you can work in the stack again.

There is one more global property, `numberFormat`. See chapter 5, "Simple Math," for a discussion of it. Table 18-3 summarizes the global properties.

Window Properties

HyperCard has four window entities: the Message box, the tool window, the pattern window, and the card window. For the Macintosh Plus and Macintosh SE built-in monitors, the card window corresponds to the full screen, but on the large Macintosh II monitors, the card window only occupies part of the screen. Each of these windows has three primary properties: the `location` (the `loc`), the `rectangle` (the `rect`), and the `visible`. We mention these properties in chapter 16, "Simulation." Unlike standard Macintosh windows, these windows cannot be resized. In addition to these basic properties, Version 1.2 supplies several auxiliary properties related to the `rectangle`.

Table 18-3. Summary of Global Properties

Syntax	Values	Comments
the blindTyping	true or false	
the cursor	1-4 (pre-Version 1.2) ibeam, plus, cross, watch, none, busy, arrow, or hand	Reset by idle, not readable
the dragSpeed	integer pixels/second	Reset by idle, 0 is maximum speed
the editBkgnd	true or false	
the language		Language names
the lockMessages	true or false	Reset by idle
the lockRecent	true or false	Reset by idle
the lockScreen	true or false	Reset by idle
the numberFormat	"...0.00..." or "...0.##..."	Reset by idle (See chapter 7, "Mouse")
the powerKeys	true or false	
the userLevel	1-5	
the userModify	true or false	New with Version 1.2, reset on stack change

By using these properties, you can position windows where you want them, control their visibilities, and position other objects relative to the windows.

Using *the* with Message Box

The various global properties require the use of the word `the`, as in `the userLevel` and `the lockScreen`. With the window properties, however, `the` is optional; you can say `the loc of tool window` or just `loc of tool window`. The reason for the discrepancy is that with window properties, you always specify which window by using the keyword `of`. The presence of this keyword informs HyperTalk that `loc` is a property and not a variable. With global properties, there is no `of`, so `the` is needed to identify the expression as being a property.

The *location* Property

The `location` property (`loc`, for short) for windows works much like the location properties of buttons and fields. The only difference is that `the location` reports the coordinates of the center of a button or field, but it reports the location of the upper-left corner of a window. In the cases of the Message box, the tool window, and the pattern window, the grab bar atop the window is not counted as part of the window. The grab bar adds 10 extra pixels to the top of these windows. See Figure 18-5. You can use the `location` property to obtain the current values, and you can use the `set` command to reposition the windows.

Example 19:

1. Go to your HyperTalk Bible stack and open the Message box.

2. Enter

```
the loc of the Message box
```

Result:

Unless you relocated the Message box, it displays `22,300`.

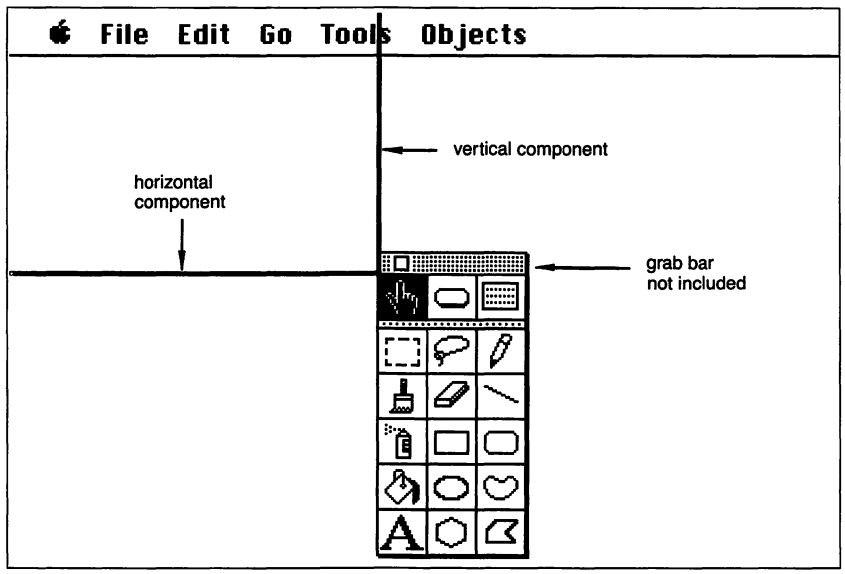


Figure 18-5. The `location` property.

This means that the upper-left corner (not counting the grab bar) is 22 pixels to the right of the left edge of the card and 300 pixels below the top of the card. The `loc` property measures distances from the upper-left corner of the card except when applied to the card window. In that case, it measures from the upper-left corner of the screen. With the small-screen Macs, the two normally are the same, but they can be different on a Macintosh II. Even with the small Macs, you can reposition the card window.

Example 20:

1. Go to your Help stack and open the Message box.
2. Enter

```
set the loc of card window to 256,171
```

Result:

The card is moved so that the upper-left corner of the card is in the middle of the screen (on a small-screen Macintosh).

3. Enter in the Message box

```
the loc of the Message box
```

Result:

The Message box displays `-234,129`.

The card corner is measured relative to the screen, but the Message box is measured relative to the card corner. Because the card corner is to the right of the left edge of the Message box, the Message box winds up with a negative horizontal coordinate. Also note that the Menu bar is unaffected.

Can we use negative coordinates for the card window?

4. Enter in the Message box

```
set the loc of card window to -256,-174
```

Result:

The lower-right corner of the card appears midscreen (on small-screen Macs), as shown in Figure 18-6.

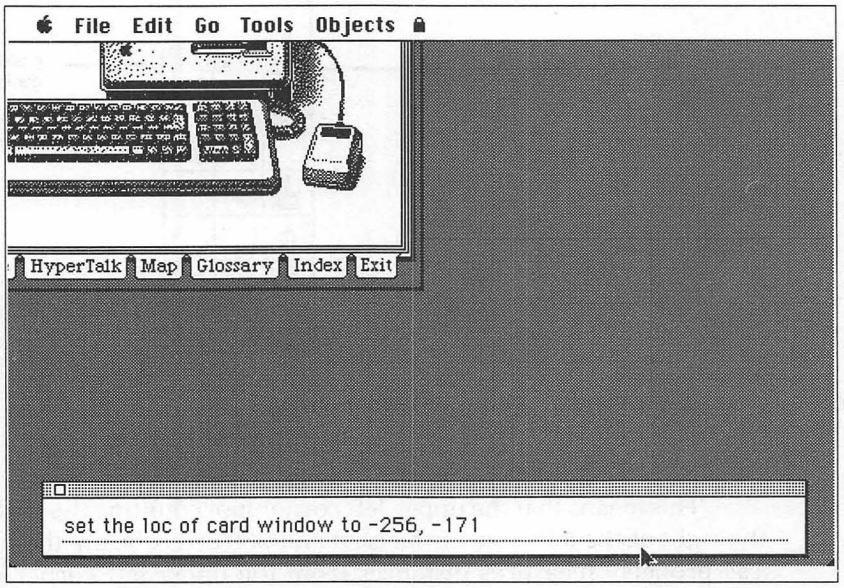


Figure 18-6. Moving a card window.

Is there anything more you should know? Yes.

5. Go to your Home stack.

Result:

The new card also is off-center!

Once you move the card window, it stays moved until you move it again or until you quit HyperCard. So enter `set card window to 0,0` to restore the proper view.

Hyper Warning

Properties Require *of*

When you write a statement that uses button, card, background, stack, or window properties, you must use the word `of`. Thus, you can say `put the hilite of cd btn5` or even `put hilite of cd btn 5` (`the` is optional). You

can't say `put the hilite cd btn 5`. In the case of a paint property, you must use the word `the` because there is no `of` to activate the property. You can say `put the lockRecent`, but you can't say `put lockRecent`.

You also can use the `location` property with the tool and pattern windows.

Example 21:

1. Go to your HyperTalk Bible stack, tear off the tool window, and locate it near the middle of the card.
2. Open the Message box.
3. Enter

```
the loc of tool window
```

Result:

The Message box displays `197,117`—or thereabouts, depending on the exact positioning.

4. Close the tool window and repeat Step 3.

Result:

The Message box displays the same coordinates it did before.

Making the tool or pattern windows disappear doesn't change their locations. You can use the `location` in scripts to position the windows in particular locations when, say, a card is opened.

Using *the* with Message Box

You can use `the Message box` or `Message box` or `message` or even just `msg` in conjunction with the `location` property, but you are restricted to `tool window`, `pattern window`, and `card window`—no `the`s allowed.

The *rectangle* Property

The `rectangle` (`rect`, for short) property returns the location of all four sides of a window, telling you both its position and its size. This property consists of four numbers: the horizontal coordinate of the left side, the vertical component of the top side, the horizontal coordinate of the right side, and the vertical component of the bottom side. See Figure 18-7. More precisely, the top and the left values are just inside the rectangle, and the bottom and right values are just outside the rectangle. For example, `the rect of the card window` is 0,0,512,342. That means the point 0,0 is in the card window, but the point 512,342 is just outside the card window.

For fields and buttons, the `rectangle` property also is settable so that you can resize those objects. For windows, however, you can't set the `rectangle` property. That's because the four windows all have a fixed size, so resizing is not allowed.

As with the `location` property, the `rectangle` property coordinates do not include the grab bar atop the Message box, the tool window, and the pattern window.

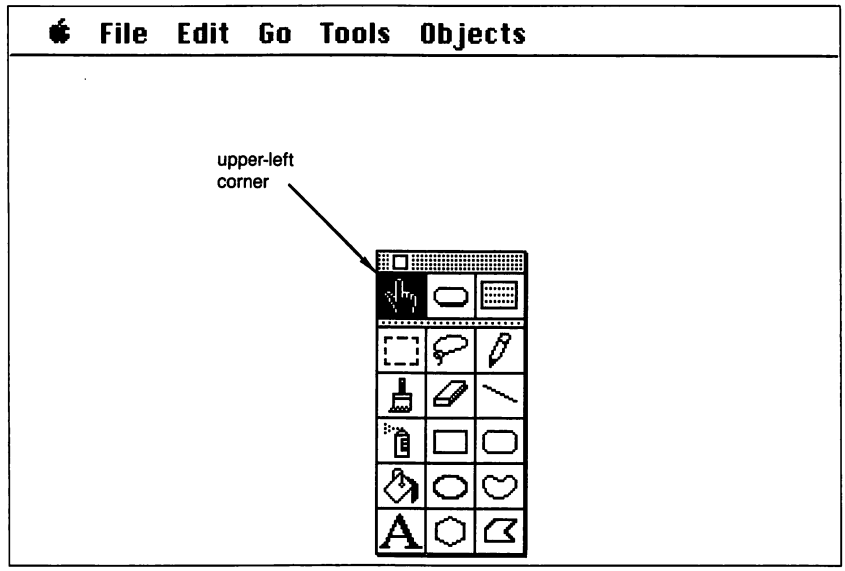


Figure 18-7. The `rectangle` property.

If you want to locate, say, the pattern window immediately below the tool window, you can use `the rect of pattern window` to determine where to put the tool window.

Example 22:

Go to your HyperTalk Bible stack, open the Message box, and enter

```
the rect of the Message box
```

Result:

The Message box displays `22,300,494,335`.

Note that the first two items correspond to the `location` property.

the screenRect Function

Version 1.2

The function `the screenRect` returns the coordinates of the corners of the current screen. The syntax is shown in Figure 18-8.

In essence, `the screenRect` works like the `rectangle` property. However, the screen is a hardware element rather than a HyperTalk software construct (like a field or the card window), so assigning a HyperTalk property to the screen doesn't make sense. Also, the fact that `the screenRect` is a function rather than a property emphasizes that you can read the screen size but not set it.

```
the screenRect
screenRect()
```

Figure 18-8. The `screenrect` function syntax.

Example 23:

1. Go to your HyperTalk Bible stack and open the Message box.
2. Enter

```
the screenRect in the Message box
```

Result:

The Message box displays `0,0,512,342` if you have a standard Macintosh Plus or SE.

Again, the last two numbers lie just outside the actual boundaries of the screen. The Macintosh Plus and SE have screens that are 512 by 342. Because the coordinates start at 0, this implies a horizontal range of 0-511 and a vertical range of 0-341.

The *visible* Property

All four windows can be hidden and revealed using the `hide` and `show` commands in the Message box or in a script. For example, `hide the message box` makes the Message box disappear, and `show the message box` brings it back. The `visible` property reports on the current visibility of the window. It's set to true if the window is visible and to false otherwise. For example, the `visible of tool window` is false if the window is hidden. You also can use the `set` command rather than `hide` and `show`. For instance, `set the visible of the message box to true` brings it into view, just as `show the message box` would.

When would you use this property? Suppose that you wanted a Message box to be present whenever a certain card was on-screen and that you wanted to restore the previous visibility state when you leave a card. The next example shows how:

Example 24:

1. Go to the Global card of your HyperTalk Bible stack.
2. Open the stack script and insert the following handlers:

```
on openCard
  global mesgstate
  put the visible of the message box into mesgstate
  set the visible of the message box to true
end openCard

on closeCard
  global mesgstate
```

continued

```
    if mesgstate <> empty
    then set the visible of the message box to mesgstate
end closeCard
```

3. Leave the card, close the Message box (if open), then return to the card.

Result:

The Message box appears.

4. Leave the card again.

Result:

The Message box disappears.

It's okay to set the `visible` property to true even if it's already true. That's why we can use `set the visible of the message box to true` without using an `if` command to check the status.

More *rectangle* Properties

Version 1.2 of HyperCard makes it easier to access individual items of the rect property. Table 18-4 summarizes these additions:

Table 18-4. Version 1.2 Enhancements to the `rectangle` Property

Property	Equivalent	Settable for windows
left	item 1 of rect	Yes
right	item 3 of rect	Yes
top	item 2 of rect	Yes
bottom	item 4 of rect	Yes
topLeft	item 1 of rect,item 2 of rect	Yes
bottomRight or botRight	item 3 of rect,item 4 of rect	Yes
height	item 4 of rect—item 2 of rect	No
width	item 3 of rect—item 1 of rect	No

Note that some properties are settable and some aren't. The rule of thumb is that if the property relates to the size of a window, such as `width`, it can't be set, but if it involves one edge or corner, it can be set. If a property is settable, the effect of setting it is to change the particular coordinate(s) mentioned.

Example 25:

1. Go to your HyperTalk Bible stack and tear off the tool window, placing it near the middle of the card.
2. Open the Message box and enter

```
the right of tool window
```

Result:

The Message box displays 270, or something similar, depending on the exact position.

3. Enter

```
set the right of tool window to 400
```

Result:

The tool window moves to the right. Its vertical position is unaffected.

Similarly, you can refer to `the left of tool window`, `the bottom of tool window`, and `the top of tool window`. Each corresponds to a single number.

4. Enter

```
the bottomRight of tool window
```

Result:

The Message box displays 400,240, or something similar.

5. Enter in the Message box

```
set the bottomRight of tool window to topLeft of tool window
```

Result:

The tool window moves so that its bottom-right corner is positioned over the previous location of the top-left corner.

You can use the width and height properties to get dimensions.

6. Enter in the Message box

```
the width of tool window
```

Result:

The Message box shows 68.

Of course, you also can use these properties in scripts. Here's one that switches the tool and pattern windows around.

Example 26:

1. Go to the Global card of your HyperTalk Bible stack and create a button called `Switch Windows`.
2. Open the button script and insert this handler:

```
on mouseUp -- Version 1.2 example
  if the visible of tool window is false then set the visible of¬
   tool window to true
  if the visible of pattern window is false then set the visible¬
   of pattern window to true
  put the topLeft of pattern window into position
  set the topLeft of pattern window to the topLeft of tool window
  play boing tempo 200 c d
  set the topLeft of tool window to position
end mouseUp
```

3. Click the Switch Windows button.

Result:

If not already visible, the windows appear and then switch positions.

Note that the `topLeft` property really is equivalent to the `location` property; both are the coordinates of the upper-left corner.

You also can use this technique with other objects, such as buttons, having these properties.

Table 18-5 summarizes the window properties.

Table 18-5. Summary of Window Properties

Syntax	Values	Comments
[the] loc[ation] of ⟨window⟩	x,y of upper-left corner of window	
[the] rect[angle] of ⟨window⟩	x1,y1,x2,y2 coordinates of corners	Not settable
[the] visible of ⟨window⟩	true or false	
New with Version 1.2		
[the] bottom of ⟨window⟩	y2 of bottom side	
[the] bot[tom]Right of ⟨window⟩	x2,y2 of bottom right corner	
[the] height of ⟨window⟩	height of window	Not settable
[the] left of ⟨window⟩	x1 of left side	
[the] right of ⟨window⟩	x2 of right side	
[the] top of ⟨window⟩	y1 of top side	
[the] topLeft of ⟨window⟩	x1,y1 of top left corner	
[the] width of ⟨window⟩	width of window	Not settable

Summary Script

Now, that you've seen how several properties work individually, let's look at how you can use a few of them together. The next example creates a background frame and draws a figure in the foreground. The drawing, at least on a Macintosh Plus, takes a while, so we use `lockScreen` to hide the work in progress. We display the watch cursor to indicate something is going on. For Version 1.2 and later, we switch to a rotating beachball cursor once the repeat loop starts.

Example 27:

1. Go to your HyperTalk Bible stack and create a new background.
2. Create a background button called `Lissajous` and give it this script:

```
on mouseUp -- Lissajous button
  set the cursor to 4 -- the watch cursor
  hide menubar
  set the visible of the message box to false
  hide background button "Lissajous"
  set the lockScreen to true
  set the editBkgnd to true
  choose rectangle tool
  drag from 0,0 to 512,342
```

continued

```
    choose round rectangle tool
    drag from 12,12 to 500, 330
    choose oval tool
    drag from 15,15 to 497,327
    choose bucket tool
    set the pattern to 9
    click at 6,6
    set the pattern to 17
    click at 20,20
    set the editBkgnd to false
    choose brush tool
    set the pattern to 12
    put 456 into x0
    put 171 into y0
    repeat with n = 1 to 48
      put round (256 + 200 * cos (pi*n/24)) into x
      put round (171 + 100 * sin(pi*n/12)) into y
      drag from x0,y0 to x,y
      put x into x0
      put y into y0
      if the version >= 1.2 and n mod 5 is 0
      then set the cursor to busy -- beach ball cursor
    end repeat
    choose bucket tool
    set the pattern to 18
    click at 200,171
    click at 300,171
    set the lockScreen to false
    choose browse tool
  end mouseUp
```

3. Click the Lissajous button.

Result:

The watch cursor appears and the screen locks. Eventually, the browse tool returns and the new appearance of the card is unveiled. See Figure 18-9.

The figure is one particular case of a class of figures called Lissajous figures after the mathematician who first described them. These figures are produced by controlling the horizontal position with a cosine function and the vertical position with a sine function. To get a Lissajous figure, the arguments of the two functions should be in the ratios of small integers. Here, for example, we use `pi*n/100` for the cosine argument and `pi*n/50` for the sine argument. This makes the sine part (the vertical component) change twice as rapidly as the horizontal component, producing the bowtie effect. A 1:1 ratio produces an oval, and other ratios, such as 3:1 or 3:2, produce other figures.

One disadvantage of using the rotating beach ball cursor is that it robs time from the other processes. To reduce that, we used the `mod` operator to cause the cursor to be updated every fifth cycle of the loop. Part of the time gained doing that, however, is lost again by the necessity of doing the `if` test every cycle.

Incidentally, this program leaves the Menu bar turned off. To get it back, type Command-M to get the Message box, then enter `show menuBar`.

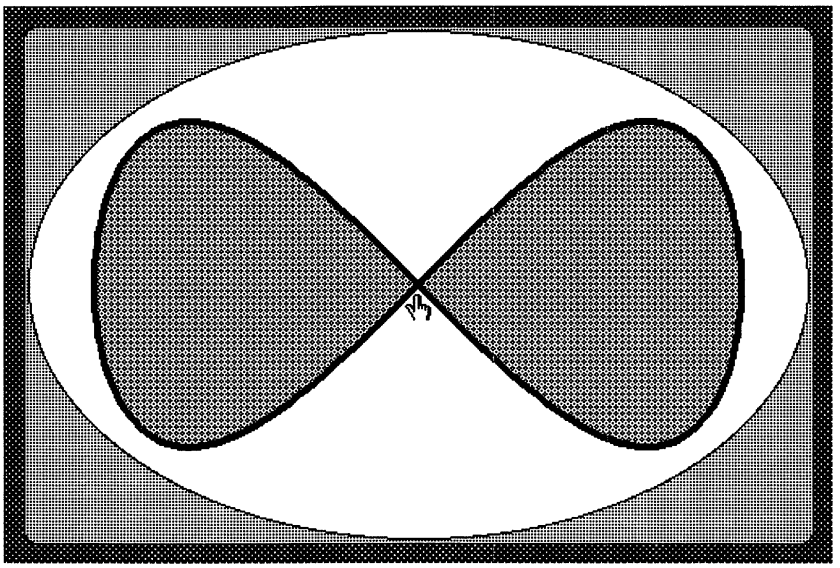


Figure 18-9. A Lissajous figure.

Review Questions

1. Which properties describe the following aspects?

   a. typing messages when the Message box is hidden
   b. whether you are in the background or on the card level
   c. whether the screen is updated
   d. the location of the Message box
   e. which powers are granted to the user
   f. whether a user can temporarily alter parts of a write-protected card

2. Which one(s) of the following properties are reset by the `idle` message?

   a. `blindTyping`
   b. `dragSpeed`
   c. `visible`
   d. `lockScreen`
   e. `cursor`
   f. `lockMessages`

3. What's wrong with each of the following?

   a. `loc of the tool window`
   b. `set userLevel to 3`
   c. `set the cursor to 6`
   d. `the upperLeft of the message box`

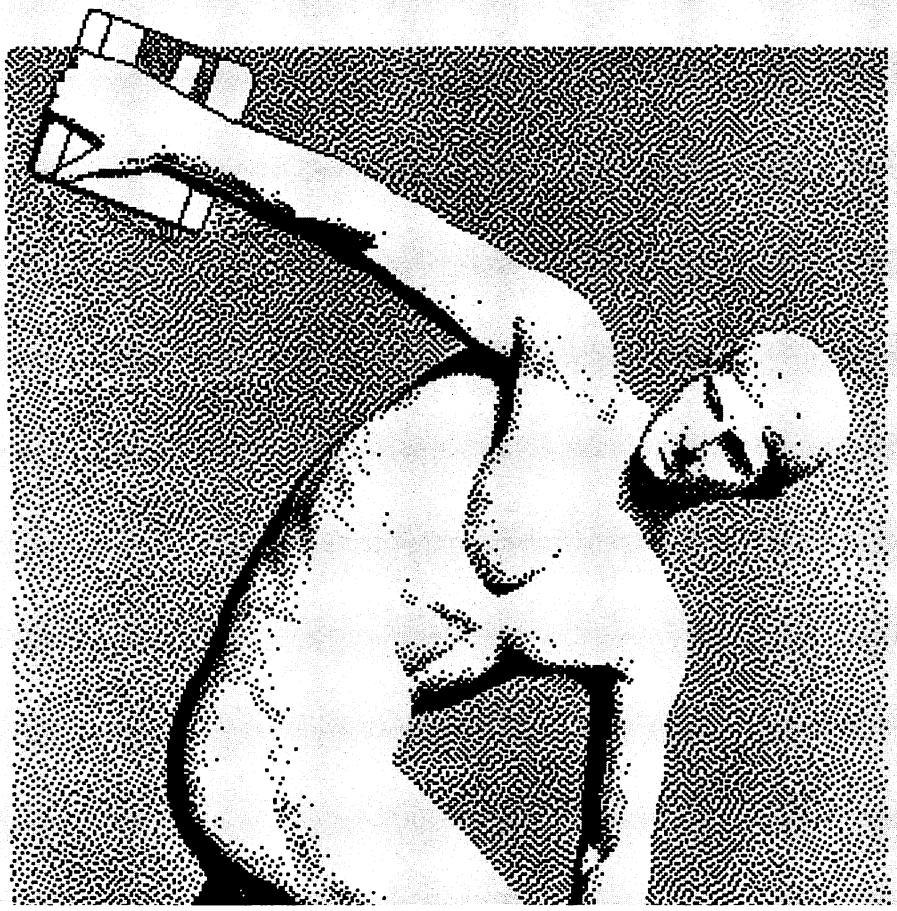
4. What effect does setting the `powerKeys` property to true have on text modes?
5. Which properties in this chapter cannot be used with a `get` command?
6. Which properties in this chapter cannot be used with a `set` command?

Exercises

1. Create an openCard handler that sets the user level to Painting, places the user into the background, enables the power keys, and places the tool and pattern windows into view along the right side of the screen.
2. Create a button that clears both the background and card pictures when clicked.
3. Create a button that draws a line across the screen, randomly changing its drawing speed as it goes.
4. Create a button that keeps showing all the cursors in succession until you press the option key.

Chapter 19

File Input and Output



Quick View

Commands

`open file`
`write to file`
`read from file`
`close file`

Concepts

Sequential-only files
Opening files
HFS and pathnames
Writing to files
Delimiters

Concepts *cont'd*

Read data from files
The FIFO system
Closing files
Generalizing file I/O
Storing chunks with multiple fields
Real-world file I/O examples
Word-processed file to HyperCard fields
HyperCard fields to word-processed file

Chapter 19

File Input and Output

Die-hard HyperCard users like to think that the Macintosh world ends at HyperCard's borders. It doesn't. There are copious amounts of information outside HyperCard in the form of text, numbers, and graphics. The problem is getting this data into HyperCard and getting HyperCard's data out to the applications beyond the pale. To input data into HyperCard from other applications or to output data from HyperCard to other applications, you must use one of two principle data-transporting routes: the Clipboard or disk files. The most obvious route is through the Clipboard with the `cut`, `copy`, and `paste` commands. The Clipboard allows you to transfer text and graphics, although the amounts that can be transfered are limited.

The other route, and this is for moving text only, is through ASCII (American Standard Code for Information Interchange) files. (Graphics can be moved with MacPaint files using the Import Paint and Export Paint menu commands.) In this chapter, we tell you all about these files and how to use them in your HyperCard scripts. We also give you some general purpose scripts for extracting data from existing stacks and putting it into files where it can be accessed by other programs. Because it is hard to do anything interesting with files without using all four of the File I/O commands, we give you a quick introduction to these commands first and then work through some examples that expand on this introduction.

Working with Files or . . . I Don't Think We're in Kansas Anymore

Files are like containers from the dark ages of computing. There is one HyperTalk command, `write`, for putting data into the file and another, `read`, for taking the data out again. For those of you who have used other computer languages, HyperTalk's files are sequential (HyperTalk does not support random-access files). Sequential files are like audio tapes; to get to the third song, or third chunk of data, you have to go past the first two songs. Unfortunately, it is not easy in HyperTalk to add data to an existing file. To do so, you have to read the existing data file into a container, such as a field or a variable, append the new data to the old with the `put` command, and

then write it back into the file. Note that this is just like the procedure we used in chapter 16, "Simulation," to change an object's script, so it shouldn't be completely new to you.

Opening Files

Before a file may be read, it must first be "opened." It's as if each file had a door that has to be opened before you can see its contents. The HyperTalk command `open file` does one of two things: It opens an existing file, or if a file doesn't already exist, it creates one and then opens it. Figure 19-1 shows the syntax for the `open file` command

```
open file <file name>
```

Figure 19-1. The `open file` command syntax.

This command probably seems pretty simple, but there is a major complication that we need to explain before we go any further. How does HyperCard know whether a file exists or not? What HyperCard does, if you tell it to `open file "SampleFile"`, is to look around on the disk for something named `SampleFile`. If `SampleFile` is on another disk, or in another folder of the same disk, HyperCard won't find it and goes ahead, in a most overbearing manner, and creates another `SampleFile` in the root volume (where HyperCard is). Because this newly created file is empty, trying to read data from it has a distinctly disappointing result. The way to avoid this sad trail of events is to specify a pathname to the file you want

The pathname is used for HFS (hierarchical file system) navigation. Like HyperCard's object hierarchy, the file hierarchy reflects a built-in order of dominance. For a desktop view correlating HFS structure with pathname, see Figure 19-2.

As usual, you don't have to type `open file "This Disk:Outer Folder:Inner Folder:SampleFile"` each time you need to specify a file. You can put the entire pathname into the variable `fileName` or into `field 2` and then use those or any other containers with the `open file` command (`open file fileName`).

HyperCard File Formats

HyperCard's internal file formats are proprietary to Apple Computer and will not be published. Apple recommends that you use HyperTalk's file commands, described in this chapter, to read formatted data.

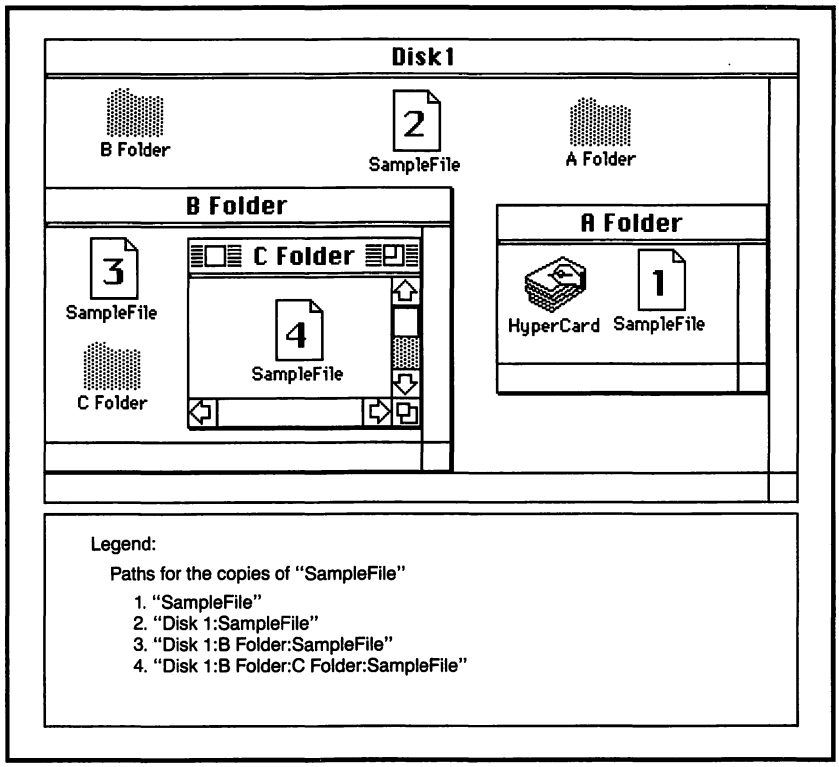


Figure 19-2. Path samples.

Writing Data to Files

Once you've opened a file, you can put data into it with the `write` command. The syntax for this command is shown in Figure 19-3.

```
write <source> to file <file name>
```

Figure 19-3. The `write` command syntax.

The source can be a quoted string expression, like `"Hello"` or `"Hello" & return`, or it can be a container, like the variable `it` or `card field 2`.

Each `write` command in HyperTalk sends one unit of data, be it a single character, a number in a variable, or several pages of text in a field, to the specified file. As long as the file is open, additional units of information can be added to the file. Often, you will want to organize your data so that the various units can be divided up again later when the file is read. HyperTalk provides a way to do exactly that through the use of delimiter characters.

Delimiters

You can place special characters between units of data to act as dividers of the units. This allows you to manipulate a data file more exactly. These special characters are called *delimiters*. The most common delimiting characters are the carriage return and the tab (called `return` and `tab` in HyperTalk). Word processor text files usually have carriage returns either at the end of each line or at the end of a paragraph; spreadsheets and database applications separate the contents of cells or fields with tabs and separate records with carriage returns. HyperTalk can handle both of these delimiting characters and any other ASCII character as well. The delimiter can be included as part of the text in the `write` command, as in `write "Hello" & return to file "fileNum"`, or it can follow the unit of data in a separate `write` command:

```
write "Hello" to file "fileNum"
write return to file "fileNum"
```

See Figure 19-4 for a sampling of delimiting characters and the results of their use with the `read` command on a sample body of text.

Reading Data to Files

To get data out of a file, you need the `read` command. The syntax for the `read` command is shown in Figure 19-5.

At first glance, there would seem to be something missing with this command. It's clear that the data is coming from `<filename>`, and it's clear that the delimiting character fits in at `<delimiter>`, but where is the data read from the source file put? As with the `get`, `ask`, and `answer` commands, `read` puts its result into the HyperTalk variable `it`.

Files are unique in HyperCard because they are the only FIFO storage system. FIFO stands for *first in first out* and is one approach to accounting, in contrast to LIFO or *last in first out*,. The stacks you've used with the `push card` and `pop card` commands, with `go back` to store cards, and to navigate around HyperCard are LIFO systems. In a FIFO file, the first unit of data you read from a file is the first one you

```
space          tab          return
   comma
This•land,•is•Your•land.→and•that•land•is•¶
My•land.•This•land•was•made•□•for•us•¶
period         shift + option "b"
```

Command	Result
read from file "Test" to return	This land, is Your land and that land is
read from file "test" to tab	This land, is Your land
read from file "test" to space	This
read from file "test" to "."	This land, is Your land and that land is My land.
read from file "test" to ","	This land,
read from file "test" to "□" (Shift-Option-b)	This land, is Your land and that land is My land. This land was made□
read from file "test" to "t"	This land, is Your land and t
read from file "test" to "T"	T
read from file "test" to "that	This land, is Your land and t

Figure 19-4. Results of `read` command with different delimiters.

```
read from file <file name> until <delimiter>
```

Figure 19-5. The `read` command syntax.

wrote in. Subsequent `read` commands pull out the units following the first until the file is empty or until you close the file. Neither FIFO nor LIFO is inherently better. You just have to remember that when you access the push/pop stack, you get the last card you put in, whereas when you access a text file, you get the first unit of information you wrote to it, as shown in Figure 19-6.

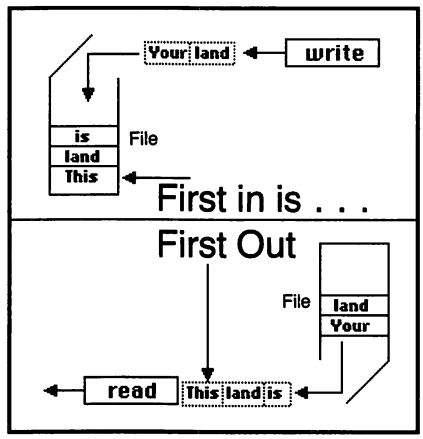


Figure 19-6. Sequential files illustrated.

Closing Files

After you've read from or written to a file, you need to close it. HyperTalk closes the file automatically when you quit, but it is good practice to close files when you are finished with them. One reason to do this is to avoid the error message produced by attempting to open an already open file. The HyperTalk `close file` command is used to close a designated file. The syntax of this command is shown in Figure 19-7.

```
close file <file name>
```

Figure 19-7. The `close` command syntax.

If the file is not in the same folder HyperCard is in, the `close file` command requires the same pathname information that `open file` needs.

In most cases you have put the file name, with or without pathname, into a container at the beginning of the file manipulation script and just reuse the same container to close the file.

You now have seen all the file I/O Commands and should have a fair idea of how files work: They must be opened by name, written to and read from by name with delimiters to keep the data organized, and closed by name.

The Plot Thickens

Think how dull life, literature, and programming would be if everything was as simple as it first appeared. Fortunately, it never quite is. HyperTalk files turn out to be modal. By this, we mean that although you can either write to or read from a file when it is first opened, after you have performed either of these actions, the situation changes and your options are more restricted.

If you begin by reading from the file, you enter a read next or append mode. You can keep reading more text until you reach the end of the file, but you can't reread the text you already read. You can also write to the file, but what you write is added to the end of what you previously read.

After you write anything to a file, you enter the write mode and can only write more text, which is appended to the end of the previously written (or read) text. Once in the write mode, the only way to get out is to close the file and reopen it.

If this sounds confusing, it's because it is. Fortunately, once you understand the modes, you will find that working with text files is really pretty simple—too simple in fact. Random access files are even trickier, but because HyperTalk doesn't offer them, you get neither the benefits nor the problems they provide.

Simple File I/O Examples

The first example is about as simple as a file example can be. We create and open a file, write a string value to it, close it, reopen it again, read the string out of the file into the variable `it`, and put the string value into the Message box.

Example 1:

1. Open the last card of your HyperTalk Bible stack, make a new card, and name it `File I/O`.
2. Make a new card button and name it `Simple File I/O`.
3. Enter the following handler in the button:

```
on mouseUp -- Simple File I/O button
  open file "SampleFile"
  write "This is a test of the file system." to file ¬
  "SampleFile"
  close file "SampleFile"

  open file "SampleFile"
  read from file "SampleFile" until tab
  put it into message box
  close file "SampleFile"
end mouseUp
```

4. Click the Simple File I/O button.

Result:

Puts the quoted string into the file and then reads the file into the Message box.

Writing to a Sample File

The next example is a reprise of the first, but this time the string is written to the file in two chunks with a tab delimiter in between so that we can separate the two chunks later.

Example 2:

1. Open the File I/O card of your HyperTalk Bible stack.
2. Make a new card button in your HyperTalk Bible stack and name it `File Sample`.
3. Type the following handler into the button:

```
on mouseUp -- File Sample button
  open file "SampleFile"
  write "This is a test" & tab to file "SampleFile"
  write "of the file system." to file "SampleFile"
  close file "SampleFile"

  open file "SampleFile"
  read from file "SampleFile" until tab
  put it into message box
  put "-- a tab -- " after the message box -- note 5 spaces
  read from file "SampleFile" until tab
  put it after message box
  close file "SampleFile"
end mouseUp
```

4. Click the File Sample button.

Result:

This handler puts the quoted strings into the file with a tab between them, reads them back one at a time, and puts them into the Message box with a dividing string in between.

Note that in this and all the examples that follow, the same file is used to store data. In each case, the previous contents of the file are replaced with the new contents.

Generalizing File I/O

The next example begins the process of generalizing the script by putting the important values, like the filename and the delimiter character, into variables where they can be easily changed. By generalizing, we mean taking a script that works for one specific case and making it apply to many different cases. For instance, by adding an `ask` command that lets the user type a filename and then putting that filename into a variable, we can use the same script to access any file. This script also uses a loop to put several units of data into the file in an automated fashion.

Example 3:

1. Open the File I/O card of your HyperTalk Bible stack.

2. Make a new card button in your HyperTalk Bible stack and name it `Variables&Random`.
3. Enter the following handler in the button:

```
on mouseUp -- Variables&Random button
  put "SampleFile" into fileName
  put tab into delimiter
  open file fileName
  repeat 5
    write random(99) & delimiter to file fileName
  end repeat
  close file fileName

  put empty into message box

  open file fileName
  repeat 5
    read from file fileName until delimiter
    put it after message box
  end repeat
  close file fileName
end mouseUp
```

4. Click the Variables&Random button.

Result:

This puts 5 random numbers into a file and displays those numbers in the Message box with a tab separating each of them (see Figure 19-8).

What happens if the data goes into the file without delimiters? Unless it is your intention to concatenate your data into a monolithic mass, you probably don't want to leave out delimiters. If you just want to save data in the most uncomplicated way, however, you can write it without delimiters and then browse through it with a word processor. All word processors have a text-file-reading capability.

A More Rigorous Example

Most file reading scripts store the file's contents in a container, so in the next example, we create a new field to display the contents of our `SampleFile` file.

Example 4:

1. Open the File I/O card of your HyperTalk Bible stack.
2. Make a new card button in your HyperTalk Bible stack and name it `File to Fields`.
3. Enter the following handler in the button:

```
on mouseUp -- File to Fields button
    put "SampleFile" into fileName
    put tab into delimiter
    open file fileName
    repeat 5
```

continued

```
      write random(99) & delimiter to file fileName
    end repeat
    close file fileName

    open file fileName
    doMenu "New Field"
    put the number of card fields into which -- line 11
    set style of card field which to rectangle
    set loc of card field which to 400,100
    repeat 5
      read from file fileName until delimiter
      put it after card field which
    end repeat
    close file fileName
end mouseUp
```

4. Click the File to Fields button.

Result:

The handler writes 5 random numbers to a file and then displays those numbers in a HyperCard field.

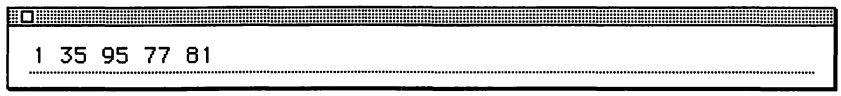


Figure 19-8. Result of Example 3.

The card field created in this handler should be the first card field on the card. Just to be on the safe side, however, in line 11, we get the number of card fields and use that number (its relative number) to refer to the field in the remainder of the script.

Storing Chunks with Multiple Fields

In the final example in this section, we take still another step into the real world by automating the creation of a field for each unit of information in the file and then reading the data into these fields. This is what you need to do if you are importing data from a spreadsheet or database into HyperCard.

Example 5:

1. Open the File I/O card of your HyperTalk Bible stack.
2. Make a new card button in your HyperTalk Bible stack and name it `Multiple Fields`.
3. Enter the following handler in the button:

```
on mouseUp -- Multiple Fields button
  put "SampleFile" into fileName
  put tab into delimiter
  open file fileName
  repeat 5
    write random(99) & delimiter to file fileName
  end repeat
  close file fileName

  open file fileName
  repeat with n = 1 to 5
    doMenu "New Field"
    put the number of card fields into which
    set rect of card field which to 60*n,30,(60*n)+50,30+16 -- 13
    set style of card field which to rectangle
    read from file fileName until delimiter
    put it into card field which
 end repeat
  close file fileName
end mouseUp
```

4. Click the Multiple Fields button.

Result:

This displays each of the 5 numbers in the file in its own field.

In this handler, line 13 positions each field on the screen by setting the `rectangle` property. The left border is 60 for the first field, 120 for the second, and 60 times the number of fields created for the others. The right side of each field is the value of the left side plus 50, so the width of each field is 50. The top of each field is 30; the bottom of each field is 46. We used 30+16 rather than 46 to emphasize that the total is the top plus the height of the default text. If you want to change the height of the `textHeight`, you also need to change 16 in the 30+16 expression. You might even want to change the expression to read `30+textHeight`.

Reading Word 3 Files into HyperCard

It is easy to read files formatted by Microsoft Word 3 into HyperCard fields. The following script shows how it's done. The script goes into a button, needs a scrolling card field called a *holder*, and uses the popular XCMD `FileName` (see chapter 20, "XCMDs and XFCNs") to present a standard Macintosh file open dialog box that you use to select a Word file on your disk. The script opens the file and reads the first 512 characters into the field holder. (You can increase the number of characters read up to a maximum of approximately 30,000. The maximum indicates the most a card field can hold.) These 512 characters are preceded by the full pathname of the file. Each file is inserted behind the previous file, so you can use this to build a quick scrolling field that peeks into all your Word files and helps you organize them.

```
on mouseUp
  put empty into cd fld holder
  repeat forever
    put FileName "WDBN") into filename1
    if filename1 is not empty then
      open file filename1 -- filename is the full pathname of a WORD file
      read from file filename1 for 16 -- move file markt to text len word
      read from file filename1 for 1 -- read the upper half of the len
      put chartonum( it ) * 256 into filesize -- shift up by 8 bits
      read from file filename1 for 1 -- get the lower half of the leng
      add chartonum( it ) mod 256 to filesize
      read from file filename1 for 238 -- move to start of text in file
      read from file filename1 for 512 -- read in the text
      close file filename1 -- IT now contains the imported data
      put return & return & " " & filename1 after cd fld holder
      put return & return & it after cd fld holder
    else
      exit repeat
    end if
  end repeat
  set scroll of cd fld holder to 0
end mouseUp
```

Real World File I/O Examples

The examples so far have assumed small units of data, but HyperCard can handle up to 30,000 characters of data in a field. The previous examples provide tools that you can use for working with paragraphs or lines separated by carriage returns or even unusual ASCII characters like Option-Z. You could, for example, read paragraphs from a word processor text file and put those paragraphs into background fields in your stack, or you could read text out of fields and into a file that you can then open and format with a word processor.

What if you want to take a text file divided into sections and subsections and move its contents into a stack with one section per card and with each subsection in its own scrolling field? Such a situation arises when you have written a document with standard, repeating, and predictable headings—like the reference section of a book. The next example shows how this can be done.

The first step in moving text into a stack is to determine the proper organization of the original text. In this case, we use ®, the registered character (Option-R) as the delimiter to mark the end of a section and, © the copyright character (Option-G) as the delimiter to mark the end of a subsection. Our script then reads the original text file once, looking for the delimiter and putting each section into its own file. Then we read each of these files, looking for the delimiter and moving the subsections into fields.

Example 6:

1. Add a scrolling card field to card File I/O of your HyperTalk Bible stack and name it `Text`. Paste or type ten paragraphs of text into the Text field. Type a ©

character (Option-G) at the end of every other paragraph (paragraph two, paragraph four, etc.). Type a ® character (Option-R) after the ® character at the end of paragraphs four and ten. This should give you one section of two two-paragraph subsections and one section of three two-paragraph subsections (see Figure 19-9).

```
"'One evening coming in with a candle I was startled to hear
him say a little tremulously, "I am lying here in the dark
waiting for death."
'Anything approaching the change that came over his features I
have never seen before, and hope never to see again.©
Oh, I wasn't touched.
I was fascinated.©®
It was as though a veil had been rent.
I saw on that ivory face the expression of sombre pride, of
ruthless power, of craven terror-of an intense and hopeless
despair. ©
Did he live his life again in every detail of desire, temptation,
and surrender during that supreme moment of complete
knowledge?
He cried in a whisper at some image, at some vision-he cried
out twice, a cry that was no more than a breath:©
'"The horror! The horror!"
I blew the candle out and left the cabin.'"©®
```

Figure 19-9. Original text paragraphs for Example 6 from *The Heart of Darkness* by Joseph Conrad.

2. Enter the following line in the Message box:

   ```
   open file "TextFile"
   ```

   (substitute another name if you've already used this one)
3. Enter the following line in the Message box:

   ```
   write card field "Text" to file "TextFile"
   ```

4. Enter

   ```
   close file "TextFile"
   ```

5. Make a new card button and name it `TextGetter`.
6. Enter the following handler in the button:

```
on mouseUp -- TextGetter handler to build a text database

  -- Preliminaries
  push card
```

continued

```
  put "TextFile" into targetFile
  put "©" into subDelimiter -- Option-G
  put "" into secDelimiter -- Option-R
  put 0 into sections

  -- read sections into files
  open file targetFile
  repeat forever
    read from file targetFile until secDelimiter -- read section
    put empty into last char of it -- strip delimiter
    if it is empty then exit repeat
    add 1 to sections
    put "File" & sections into fileName -- concatenate file name
    open file fileName
    write it to file fileName -- write section to separate file
    close file fileName
  end repeat
  close file targetFile

  -- read subsections into fields
  repeat with sectionCount = 1 to sections
    put 0 into subs
    put "File" & sectionCount into fileName -- concatenate file name
    open file fileName
    read from file fileName until subDelimiter -- read subsection
    if it is empty then exit repeat
    put empty into last char of it -- strip delimiter
    put it into nextField -- store subsection in variable
    doMenu "New Card"
    repeat forever
      add 1 to subs
      doMenu "New Field"
      set loc of card field subs to subs * 90,subs * 90
      set style of card field subs to scrolling
      if first char of nextField is return
      then put empty into first char of nextField -- strip return
      put nextField into card field subs -- put subsec into field
      read from file fileName until subDelimiter -- read next subsec
      if it is empty then exit repeat
      put empty into last char of it -- strip delimiter
      put it into nextField

   end repeat
    close file fileName
  end repeat
  pop card
  choose browse tool
end mouseUp
```

7. Click the TextGetter button.

Result:

The first card after your File I/O card has two scrolling card fields. The next card has three scrolling card fields with two paragraphs of text in each field (see Figure 19-10 for the card with three fields).

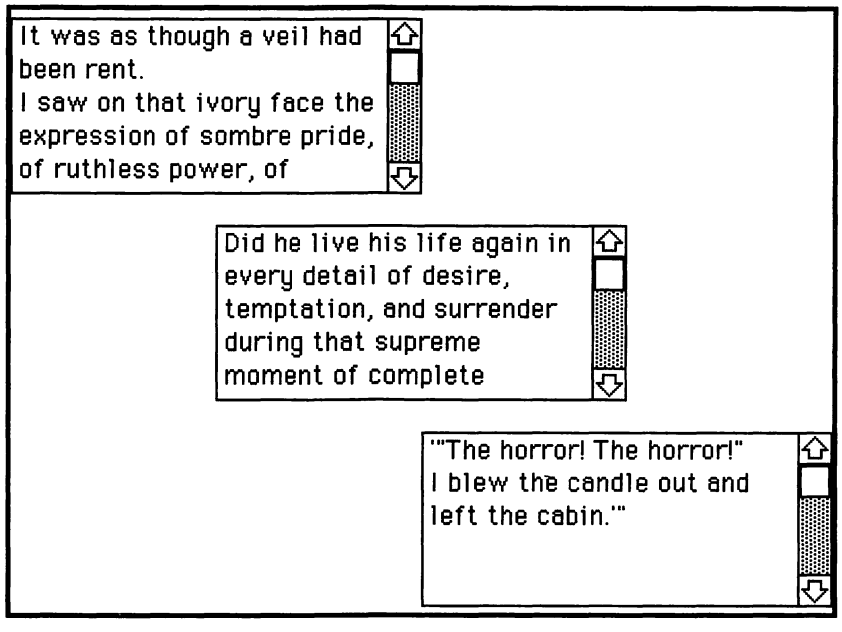


Figure 19-10. Result of Example 6, section 2.

The Preliminaries section is pretty minimal, but the rest of the script is fully fleshed out.

The next example is the reverse of the last; it goes from card to card writing fields to a file. The units of data in each field are separated with one delimiter, and the fields on different cards are separated by a second delimiter. Note that this example has the capability of writing both background and card fields to the same file.

Example 7:

1. Open the File I/O card of your HyperTalk Bible stack.
2. Add a new card button to your HyperTalk Bible stack and name it `TextPutter`.
3. Enter the following script in the button:

```
on mouseUp -- TextPutter button

  -- Preliminaries
  ask "Export to which file?" with "Disk:Folders...:File"
  if it is empty then exit mouseUp
  put it into targetFile
  answer "Card delimiter?" with "return"  or "tab" or "Other"
  if it is "tab" then
    put tab into cDelimiter
  else
    if it is "return" then
      put return into cDelimiter
    else
```

continued

```
    ask "What character?"
    if it is empty then exit mouseUp
    put it into cDelimiter
  end if
end if
Answer "Field delimiter?" with "return"  or "tab" or "Other"
if it is "tab" then
  put tab into fDelimiter
else
  if it is "return" then
    put return into fDelimiter
  else
    ask "What character?"
    if it is empty then exit mouseUp
    put it into fDelimiter
  end if
end if
ask "Starting with Card..." with "1"
if it is empty then exit mouseUp
put it into cardStart
ask "Ending with Card..." with the number of cards
if it is empty then exit mouseUp
put it into cardStop
answer "Extract data from..." with "Bkgnd Fields" or "Card Fields"¬
or "Other"
push card

-- Write just background fields
if it is "Bkgnd Fields" then
  open file targetFile
  repeat with n = cardStart to cardStop
    go card n
    repeat with p = 1 to the number of bkgnd fields
      write background field p & fDelimiter to file targetFile
    end repeat
    write cDelimiter to file targetFile
  end repeat
else

  -- Write just card fields
  if it is "Card Fields" then
    open file targetFile
    repeat with n = cardStart to cardStop
      go card n
      repeat with p = 1 to the number of card fields
        write card field p & fDelimiter to file targetFile

    end repeat
      write cDelimiter to file targetFile
    end repeat
  else
    answer "Extract data from..." with "Cancel" or "Crd & Bkgnd"
    if it is "Cancel" then exit mouseUp
```

continued

```
        -- Both card and background fields
        open file targetFile
        repeat with n = cardStart to cardStop
          go card n
          repeat with p = 1 to the number of bkgnd fields
            write background field p & fDelimiter to file targetFile
          end repeat
          repeat with p = 1 to the number of card fields
            write card field p & fDelimiter to file targetFile
          end repeat
          write cDelimiter to file targetFile
        end repeat
      end if
    end if
    close file targetFile
    pop card

    -- Results display
    open file targetFile
    doMenu "New Field"
    set style of card field the number of card fields to scrolling
    read from file targetFile until "¿" -- option + shift + /
    put it into card field the number of card fields
    choose browse tool
  end mouseUp
```

4. Click the TextPutter button.

Result:

The last section of the script puts the contents of the new file into a scrolling field where you can see both the text and the delimiting characters.

The last Results display section is not necessary but is useful for confirming the results of the script. It is also something of a bonus because it demonstrates that you can adapt this script for collecting text from fields even if you are not sending the collected data out to a file. This script has the capability of working with two delimiters, writing card fields on multiple cards, and writing both card and background fields from the same card. Here, we assumed that you will be collecting the data from every field, of whichever type, or types, you've specified.

Summary

In this chapter, you saw more about HyperTalk's text files than you probably wanted to know. Importing and exporting data is one of HyperCard's weaknesses, so the more comfortable you are with text files, the more you will be able to work around these weaknesses. You learned how to create, address, write to, and read from ASCII files. We also provided scripts to automate collecting data from a stack and exporting it to a file or for importing data from other applications into HyperCard and distributing it to a database the script creates on the fly.

From this chapter, you gained a range of tools for use in manipulating both files and databases, but you will save yourself a lot of trouble in the long run if you carefully plan how you are going to input and output your data before you start collecting it. By nature, databases tend to be highly structured, which makes them easier to work with in a script. There are usually a fixed number of fields per record and a fixed number of records per screen. It is easy to transfer data from a structured database to a HyperCard database structure that would mirror the original. The most important thing in such a situation is to get the proper delimiters into the file and to use those delimiters in the proper order with the `read` command.

Free-form databases (like books, papers, and the like) can be manipulated if you superimpose a structure based on delimiting characters. The more regular the original structure the better: 10 sections in separate fields on one card representing one chapter and only two sections on the next card looks odd, and you have to take care with the sizing and positioning of those two fields.

Although import/export is not a HyperCard strength and you probably didn't start learning HyperTalk so that you could fool with antiquated text files, after you get the knack, text file routines are fast and powerful and can be used to advantage in many HyperCard applications.

Review Questions

1. Will attempting to open an already open file generate an error?
2. What kind of files does HyperTalk support?
   a. sequential
   b. random-access
3. What do you call a character that separates one unit of data from another unit?
4. Can any character be a delimiter?
5. What is the order of file, stack, and folder names in a pathname?
6. You have a file named `1988` in folder `Taxes For` on stack `Data`. Are each of the following three pathnames valid or invalid?
   a. `data:taxes for:1988`
   b. `DATA:TAXES FOR:1988`
   c. `Data:TaxesFor:1988`
7. If you wanted to put the contents of the Message box and a tab into file `MessageFile`, what would the appropriate `write` command look like?
8. Where is the data stored following a `read from <filename> until <delimiter>` command?
9. When you open a file and read one data unit, will the first unit read be
   a. the last one written?
   b. the first one written?
10. If you open an existing file, read its contents, and then write data to it, will the new data

a. be appended to the old data?
b. replace the old data?

11. If you open an existing file and write new data to it, will the data

    a. be appended to the old data?
    b. replace the old data?

12. What's wrong with the following fragment?

```
open file "Sample"
write field 2 to file "Sample"
read from file "Sample"until return
close file "Sample"
```

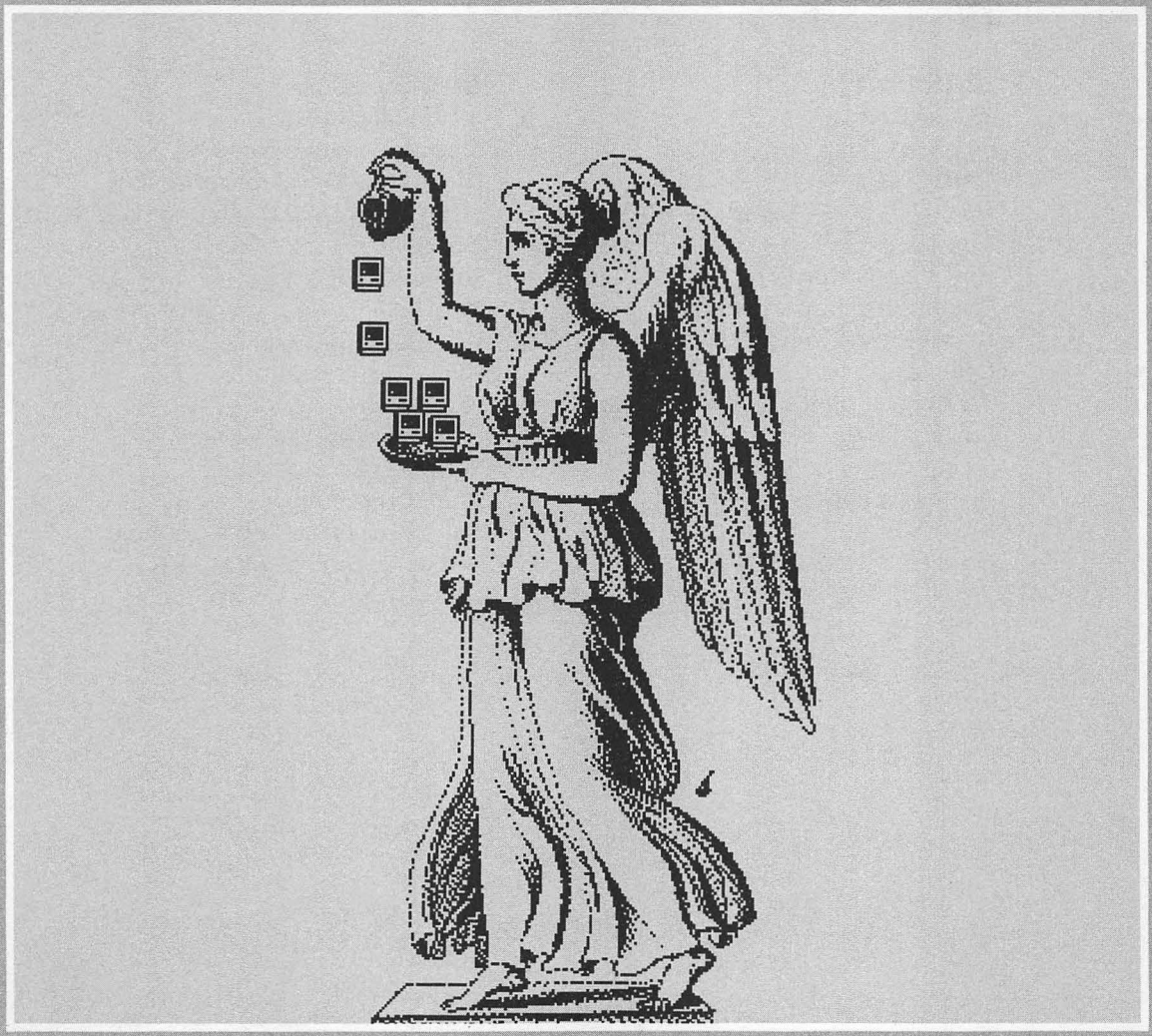
Exercise

Write a no-nonsense script to import data from a text file into a HyperCard database. Don't worry about the user interface. Hard-wire the script for one file name `ImportData`. Assume that the data is stored with tabs between fields and returns between records. Assume that the data will go into existing background fields with one record per card. The script should go into a card button on the first card and make new cards for each record.

Chapter 20

XCMDs and XFCNs

by Steve Maller



Quick View

Function
`changeCase`

Concepts
How does an XCMD-XFCN work?
The heap
How data is stored
The best of both worlds
Pascal versus HyperTalk
Callbacks
Sending commands and functions
String manipulation callbacks
String conversion callbacks
Accessing objects callbacks
General utility callbacks
Needs for writing XCMDs

Concepts *cont'd*
Inside Macintosh
Macintosh technical notes
HyperCard Developer's Toolkit
What you can and can't do with XCMD-XFCNs
Toolbox managers
`ExitToShell`
Using `SysEnvirons`
HyperCard's heap
File I/O
Using `Get/WaitNextEvent`
XCMD-XFCN's variables
The `XCmdBlock`
The `ChangeCase` XFCN example
Installing your XCMD-XFCN

Chapter 20

XCMDs and XFCNs

This chapter has three objectives: to describe the HyperCard external function and command architecture, to teach a bit of Macintosh programming, and to supply an example XFCN. In doing so, it maintains a significantly higher technical level than previous chapters. This is deliberate; Macintosh programming in general is not trivial, and more specifically, the development of XCMDs-XFCNs can be very tricky.

Here we delve into the creation of the XFCN called `ChangeCase`. `ChangeCase` converts the capitalization of text in one of four ways (all caps, all lowercase, capitalize words, or capitalize sentences). The `ChangeCase` stack is available on networks, such as CompuServe, as well as on The Waite Group's HyperTalk Bible stack, and an example script that accesses `ChangeCase` is listed in the source code. `ChangeCase` is written in Apple's MPW Pascal, but it can be easily converted to other Pascals and could be rewritten in C with only a little more effort.

Documents or other reference materials discussed in this chapter and available from the Apple Programmers' and Developers' Association are *Inside Macintosh,* "Macintosh Technical Notes" "HyperCard Developer's Toolkit," and "MacTutor." Many of these are also available from online services such as CompuServe and Delphi and from most user groups.

How Does an XCMD-XFCN Work?

An XCMD is used the same way as a user-defined command or function. You can pass up to 16 arguments to it. An XFCN is expected to return a result, and an XCMD can do so if an error occurs (using the HyperTalk function `the result`). Although the internal structure of a HyperCard container is not generally of interest to a scriptor, it is important to an XCMD-XFCN developer.

A HyperCard container is referenced from an XCMD-XFCN through its Handle. A *Handle* is a Macintosh Memory Manager data type; technically, "it is a pointer to a master pointer to a relocatable block on the heap." What does this mean?

There are two areas of memory available to a Macintosh program for storing data: the stack and the heap. The stack is used by program code to store variables

and other housekeeping information. It grows as routines call more routines and the new routines put their data on the stack, and it shrinks as those routines terminate and return to the routine that called them.

The Heap

The heap is a set size, but the blocks within it are allocated dynamically while a program is running. There are two types of blocks that can be allocated within the heap: relocatable and nonrelocatable. Relocatable blocks are nice and friendly to a running program because they can be moved to another place in the heap if memory becomes fragmented, or if space becomes too tight, they can even be removed. Normally blocks are made purgeable (removeable) only if the data contained in these blocks is backed up on disk somewhere. Nonrelocatable blocks are dangerous because they cannot be moved and should never be used except for very short periods of time. One nonrelocatable block in the wrong place can easily cause a program to crash.

The Memory Manager maintains a list of pointers (master pointers) to all the relocatable blocks in the heap. (The master pointers themselves are nonrelocatable.) This way, when the Memory Manager moves a relocatable block, it just updates the master pointer to that block. The application references a relocatable block through a Handle, which is the address of a master pointer. When a block moves, the Handle is still valid because it always knows where to go to access the actual address of the block itself. Figure 20-1A shows a Handle in RAM; Figure 20-1B shows the same handler after a heap contraction.

The advantage of this approach (commonly known as double indirection) is that the Memory Manager can move a block within the application heap when making a memory allocation. After a block is moved, its Handle still points to the same place; only the master pointer is updated to the block's new location.

How Data Is Stored

Data in a HyperTalk container is stored as a zero-terminated character array (see Figure 20-2), regardless of the data type and length. When HyperTalk reads from a string, it does so until it finds a character with an ASCII value of zero (0). HyperTalk stores all of its data— text, integers, and even real numbers—as zero-terminated strings. A zero-terminated string is unlimited in length, unlike a Pascal string which is limited to 255 characters. (Hey, you C programmers: stop smirking!)

HyperCard's Built-In XCMDs

HyperCard comes with a few XCMDs of its own. The `Flash` XCMD causes the screen to invert black to white and then back again. The number of times this should happen is fed to `Flash` as a parameter. Because `Flash` is in HyperCard, it is available to all stacks. Another not-so-well-known

XCMD in HyperCard is called `GetDocs` and can be found in the Documents stack. This XCMD appears as `getdocs line i of documents`, which passes as an argument the folder name from the Documents card in your Home stack. `GetDocs` then creates a card with that folder name. Thus, `GetDoc` supplies a way to create a card index of all your folders and files.

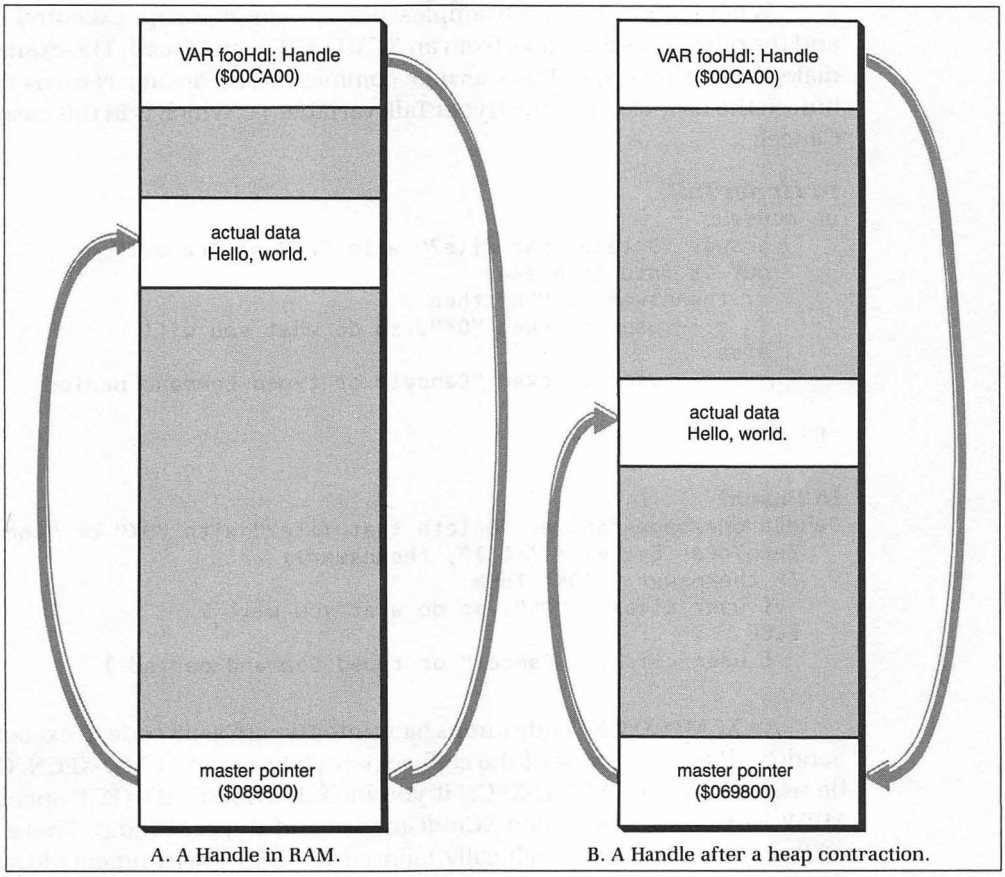


Figure 20-1. Handles on the Macintosh.

Figure 20-2. Zero-terminated strings.

The Best of Both Worlds

In addition to complete access to the Macintosh Toolbox, an XCMD-XFCN also can use all of HyperCard's features. You can do anything from an XCMD-XFCN that you can do from HyperTalk: issue requests, send messages, perform calculations, and even run another XCMD-XFCN. These communiques between your code and HyperCard's are called *callbacks*.

What follows are two examples: one is a simple script executed in HyperTalk, and the other uses callbacks from an XCMD-XFCN (in Pascal). The examples display a dialog box using HyperTalk's `answer` command. The `answer` returns the text of the button the user chose in the HyperTalk variable `it`, which is in this case either OK or Cancel.

In HyperTalk

```
on mouseUp
     answer "Delete that file?" with "OK" or "Cancel"
     put it into theAnswer
     if theAnswer is "OK" then
          -- user clicked "OK", so do what you will
     else
          -- user clicked "Cancel" or typed Command-period
     end if
end mouseUp
```

In Pascal

```
SendCardMessage('answer "Delete that file?" with "OK" or "Cancel"');
   ZeroToPas(EvalExpr('it')^, theAnswer);
   IF theAnswer = 'OK' THEN
     { user clicked "OK", so do what you will }
   ELSE
     { user clicked "Cancel" or typed Command-period }
```

An XCMD-XFCN's code jumps back into HyperCard's code to execute a callback. SendCardMessage is one of the callbacks available to an XCMD-XFCN. Callbacks can be used from your XCMD-XFCN if you include two XCMD-XFCN-specific files. The MPW Pascal files are named XCmdGlue.inc and HyperXCmd.p. There are versions of these files that are specifically tailored to other environments (e.g.: Lightspeed Pascal). Contact your development system's publisher for information on the files required for their system. Most have translated the MPW files so that they are compatible with their language.

Callbacks

Following is a list of the standard callbacks that HyperCard accepts. See the HyperCard Developer's Toolkit file HowToUse.doc for further information on these routines.

Sending Commands and Functions

`SendCardMessage(msg: Str255);` *(procedure)*

This is the same as typing the string passed in `msg` into the message box and pressing Return.

`EvalExpr(expr: Str255): Handle;` *(function)*

This evaluates the HyperTalk expression passed in `expr` and returns a handle to a zero-terminated string containing the result of the evaluation. For example, `EvalExpr('the long date')` returns a Handle to a string containing the current date in the long format (Saturday, June 25, 1988).

String Manipulation

`StringLength(strPtr: Ptr): LongInt;` *(function)*

Returns the number of characters in the zero-terminated string pointed to by `strPtr`. Note that `strPtr` is a Ptr (pointer), not a Handle. This means that the Handle to the string must be partially dereferenced in order for `StringLength` to work. For example, if `foo` is a handle to a zero-terminated string, you must pass `foo^` to StringLength.

This is similar to HyperTalk's `the number of chars in <container>` function.

`StringMatch(pattern: Str255; target: Ptr): Ptr;` *(function)*

Performs a case-insensitive search looking for `pattern` (a Pascal string) in the zero-terminated string pointed to by `target`. If the search is successful, the location of the first matching character is returned as the function result. If the search is unsuccessful, StringMatch returns `nil`.

This is similar to HyperTalk's `offset` function.

`ZeroBytes(dstPtr: Ptr; longCount: LongInt);` *(procedure)*

This sets `longCount` bytes beginning at `dstPtr` to 0. No boundary checking is done, so you should be careful with this. ZeroBytes will write past the end of a zero-terminated string, for example.

String Conversions

`PasToZero(str: Str255): Handle;` *(function)*

Converts `str` to a zero-terminated string and returns a Handle to the new string. Your XCMD-XFCN must dispose of the handle.

`ZeroToPas(zeroStr: Ptr; VAR pasStr: Str255);` *(procedure)*

Converts the zero-terminated string pointed to by `zeroStr` to a Pascal string and returns the string in `pasStr`.

`StrToLong(str: Str31): LongInt;` *(function)*

Returns `str` converted to a long (32-bit) unsigned integer. Unsigned long integers can range from 0 to 4,294,836,225.

`StrToNum(str: Str31): LongInt;` *(function)*

Returns `str` converted to a long (32-bit) signed integer. Signed long integers can range from (-2,147,418,112) to 2,147,418,112.

`StrToBool(str: Str31): BOOLEAN;` *(function)*

Returns the Pascal strings `'true'` and `'false'` as booleans (TRUE and FALSE).

`StrToExt(str: Str31): Extended;` *(function)*

Returns `str` converted to an *extended* type. Extended numbers contain a sign bit, 15 bits for the exponent and 63 bits for the significand. This is the standard data type for SANE, the Standard Apple Numeric Environment.

`LongToStr(posNum: LongInt): Str31;` *(function)*

Returns `posNum` (a 32-bit unsigned integer) as a Pascal string.

`NumToStr(num: LongInt): Str31;` *(function)*

Returns `num` (a 32-bit signed integer) as a Pascal string.

`NumToHex(num: LongInt; nDigits: INTEGER): Str31;` *(function)*

Returns `num` as a Pascal string containing a hexidecimal (base 16) representation of the value of num, expanding the string to `nDigits` in length.

`BoolToStr(bool: BOOLEAN): Str31;` *(function)*

Returns a string (`'true'` or `'false'`) representing the boolean value of `bool`.

`ExtToStr(num: Extended): Str31;` *(function)*

Returns `num` (an extended type) as a Pascal string.

Accessing Objects

`GetGlobal(globName: Str255): Handle;` *(function)*

Returns a Handle to a zero-terminated string containing a copy of the contents of the HyperTalk global container named `globName`.

`SetGlobal(globName: Str255; globValue: Handle);` *(procedure)*

Copies the zero-terminated string to which `globValue` is a Handle into the HyperTalk global container named `globName`. If `globName` doesn't yet exist, `SetGlobal` creates it.

`GetFieldByName(cardFld: BOOLEAN; fldName: Str255): Handle;` *(function)*

Returns a Handle to a zero-terminated string containing a copy of the contents of the field named `fldName`. If `cardFld` is TRUE, `fldName` is a card field; otherwise, it is a background field.

This is similar to HyperTalk's `[card/bkgnd] field fldName` function.

`GetFieldByNum(cardFld: BOOLEAN; fldNum: INTEGER): Handle;` *(function)*

Returns a Handle to a zero-terminated string containing a copy of the contents of the field with the number `fldNum`. If `cardFld` is TRUE, `fldNum` is a card field; otherwise, it is a background field.

This is similar to HyperTalk's `[card/bkgnd] field fldNum` function.

`GetFieldByID(cardFld: BOOLEAN; fldID: INTEGER): Handle;` *(function)*

Returns a Handle to a zero-terminated string containing a copy of the contents of the field with the ID `fldID`. If `cardFld` is TRUE, `fldID` is a card field; otherwise, it is a background field.

This is similar to HyperTalk's `[card/bkgnd] field ID fldID` function.

`SetFieldByName(cardFld: BOOLEAN; fldName: Str255; fldVal: Handle);` *(procedure)*

Copies the zero-terminated string to which `fldVal` is a Handle into the field named `fldName`. If `cardFld` is TRUE, `fldName` is a card field; otherwise, it is a background field. The XCMD-XFCN must dispose of the Handle afterwards.

This is similar to HyperTalk's `put fldVal into [card/bkgnd] field "fldName"` command.

`SetFieldByNum(cardFld: BOOLEAN; fldNum: INTEGER; fldVal: Handle);` *(procedure)*

Copies the zero-terminated string to which `fldVal` is a Handle into the field with the number `fldNum`. If `cardFld` is TRUE, `fldNum` is a card field; otherwise, it is a background field. The XCMD-XFCN must dispose of the Handle afterwards.

This is similar to HyperTalk's `put fldVal into [card/bkgnd] field fldNum` command.

`SetFieldByID(cardFld: BOOLEAN; fldID: INTEGER; fldVal: Handle);`

Copies the zero-terminated string to which `fldVal` is a Handle into the field with the ID `fldID`. If `cardFld` is TRUE, `fldName` is a card field; otherwise, it is a background field. The XCMD-XFCN must dispose of the Handle afterwards.

This is similar to HyperTalk's `put fldVal into [card/bkgnd] field ID fldID` command.

Neat Testing Tricks

For testing XCMD-XFCNs, a useful technique is to call your external code over and over with the statement `put the heapspace` in between the calls. If you see this value decreasing rapidly, you might suspect that your external is allocating heap space and forgetting to dispose of it.

General Utilities

`StringEqual(str1,str2: Str255): BOOLEAN;` *(function)*

Compares the two Pascal strings `str1` and `str2` (case-insensitive and diacritical-sensitive) and returns TRUE if the two strings are identical; otherwise, it returns FALSE.

`ReturnToPas(zeroStr: Ptr; VAR pasStr: Str255);` *(procedure)*

Copies characters from the zero-terminated string pointed to by `zeroStr` into the Pascal string passed in `pasStr`, stopping at the first return character (ASCII $0D), the end of the zero-terminated string, or the 255th character, whichever comes first.

This is similar to HyperTalk's `get line n of containerName` function.

`ScanToReturn(VAR scanPtr: Ptr);` *(procedure)*

Scans along the zero-terminated string pointed to by `scanPtr`, stopping just past the first return character or at the end of the string. `scanPtr` is incremented to point to the new location.

`ScanToZero(VAR scanPtr: Ptr);` *(procedure)*

Scans along the zero-terminated string pointed to by `scanPtr`, stopping at the end of the string. `scanPtr` is incremented to point to the new location.

What You Need to Get Started

The Macintosh is not a forgiving environment to master, yet those who take the time are almost always rewarded. There are a number of resources (pardon the pun!) that vary in importance from absolutely required to very helpful. The list below is not absolutely complete, but these are high quality reference works that are essential to any serious Macintosh development.

Inside Macintosh

No Macintosh developer can hope to get very far by "hacking" alone. Apple and Addison Wesley publish a five-volume set called *Inside Macintosh.* This is the definitive set of technical information on the internals of the Macintosh family. It is absolutely required for any significant development work. The volumes are divided among "managers" or functional groupings of routines. The Memory Manager, Window Manager, and Event Manager are examples of Macintosh managers. Mysterious subjects such as QuickDraw and the Memory Manager are exhaustively documented, if not demystified. Even the engineers at Apple use it; there is no higher truth!

Inside Macintosh Volumes I-III cover all of the fundamentals and take you in

your journey through the original Macintosh Toolbox (64KB ROM) functionality. (Required)

Inside Macintosh Volume IV covers all of the additional functionality in the 128KB ROM Toolbox, which includes the Macintosh 512e and Macintosh Plus. New features include the SCSI interface, the Hierarchical File System (HFS), the List Manager, Time Manager, and additions in old standbys such as QuickDraw and the Resource Manager. (Required)

Inside Macintosh Volume V introduces the Macintosh SE and Macintosh II feature sets. Countless new managers and new features to older managers took their bows in this book, all the way from the Start Manager to the ShutDown Manager, including the big one: Color! This volume is optional, yet recommended, even if you are not going to be using any SE- or II-specific routines. It is helpful to have this as a reference to some of the latest additions. You never know when you might want to dabble in the Deferred Task Manager or the New and Improved SCSI Manager!

"Macintosh Technical Notes"

"Macintosh Technical Notes" are published on a regular basis by the wizards of Apple's Developer Technical Support (DTS) group. DTS is the guardian of developers both inside and outside Apple. When all else fails, they are the ones who can help with such things as a suspected ROM bug. When "all else fails" for too many people in the same manner, it means that the mistake is Apple's and an engineer will author a Technical Note on the subject. "Macintosh Technical Notes" are distributed to all Apple Certified Developers, posted on all the networks, and sold through APDA. Apple has published over 200 of them on topics as diverse as secrets of the bundle bit to how to recover a crashed SCSI hard disk. (Required)

"HyperCard Developer's Toolkit"

This combination of documentation and software comprises all that Apple knows about XCMD-XFCN development. It includes source code in Pascal and C, interface and header files for Pascal and C development (MPW only), and many examples. (Required)

"MacTutor"

"MacTutor" is the most widely-read and respected Macintosh technical journal. Its staff and contributors are dedicated to Macintosh programming, and they offer a wide variety of topics every month in an equally wide variety of languages. Subscriptions are inexpensive, all source code is available separately on disk, and the journal publishes a compilation ("Best of MacTutor," Volume *n*) every year or so, containing the most interesting articles since the last compilation.

How to Build Your Own XCMDs-XFCNs

XCMDs-XFCNs can be written in virtually any Macintosh-compatible language that can produce native code. Contact your development system vendors; they have the necessary "hooks" to allow you to develop XCMDs-XFCNs. The Apple Programmers' and Developers' Association (A.P.D.A.) sells a toolkit containing the interfaces for Apple's MPW C and Pascal compilers. Symantec has developed versions of these interfaces for its Lightspeed Pascal and LightspeedC compilers, and Borland has done the same for its Turbo Pascal compiler.

You should also have a good debugger. Apple's Macsbug and SADE, Icom Simulation's TMON, Jasik Design's The Debugger are examples of currently-available debuggers. When things go awry in an XCMD-XFCN, it can be difficult to untangle your code. If you don't want object-code debugging, you should at least have a good heap diagnosis tool.

A.P. D.A. carries one of the best selections of development tools. Some of the more dedicated Macintosh software retailers and mail order distributors also carry good selections of development tools.

What You Can and Can't Do

These guidelines are in addition to Apple's "Macintosh Technical Note #117—Compatibility: Why and How." Those guidelines include

Check errors returned by function results
Do not access hardware directly
Do not assume that rowBytes is equal to the width of the display
Do not assume that the display is at a fixed location
Do not assume the display is a particular size
Do not create or use fake handles
Do not use NIL handles or pointers, nor pass them to the Toolbox
Do not use any fields, global variables, or bits that are reserved
Do not write code that modifies itself
Think twice about code designed strictly as copy protection

Although the preceding guidelines apply to any Macintosh developer, the following text lists some special concerns for the XCMD-XFCN developer, but, remember, all of the preceding rules still apply.

In many ways, an XCMD-XFCN is similar to a Desk Accessory: You are a guest and should act accordingly. As you can see, developing an XCMD-XFCN can be every bit as complicated as developing a complete Macintosh application.

Don't Declare Global or Static Variables
Externals can't share HyperCard's global data space. You can allocate space on the heap with `NewHandle` or `NewPtr` calls, but you must dispose of that space before returning to HyperCard.

Toolbox Managers

Don't initialize any of the Macintosh Toolbox managers (ie: `InitWindows`); that has already been done by HyperCard. There are virtually no initializations for your XCMD-XFCN to do, except possibly check for available RAM and flush the event queue if you have a dialog box or other interactive interface.

ExitToShell

Don't call `ExitToShell`. Terminate your XCMD-XFCN with an `"END."` statement and that will return to the HyperCard environment.

SysEnvirons

Don't make assumptions about the host machine; use `SysEnvirons`. This function returns a data structure containing a considerable amount of information about the CPU configuration. The original version of the call returns the following information. See the "Macintosh Technical Note #129: SysEnvirons" for more details.

environsVersion

The version of `SysEnvirons` that responds to your inquiry. Currently, this is always 1, but that will change over time as `SysEnvirons` becomes more verbose.

machineType

This will return one of the following predefined constants:

```
envMachUnknown  = 0;
env512KE        = 1;
envMacPlus      = 2;
envSE           = 3;
envMacII        = 4;
```

systemVersion

This integer field returns the System version number represented as two byte-long numbers, separated by an implied decimal point (but not a fixed point number), that is, System 6.0 returns $0600 in the field.

processor

Returns the microprocessor model.

```
envCPUUnknown   = 0;
env68000        = 1;
```

continued

```
env68010        = 2;
env68020        = 3;
```

hasFPU

This field returns TRUE if the MC68881 math coprocessor is installed.

hasColorQD

This field returns TRUE if Color QuickDraw is available. It does not imply that a color monitor is available or even if the pixel depth is greater than one. In other words, it doesn't tell you how the current display is configured. What it does tell you is that you can use Color QuickDraw routines to find out these things. Attempts to use any of these routines on a Macintosh Plus, for example, results in a system crash.

keyBoardType

Returns the model of the keyboard.

```
envUnknownKbd       = 0;
envMacKbd           = 1;
envMacAndPad        = 2;
envMacPlusKbd       = 3;
envAExtendKbd       = 4;
envStandADBKbd      = 5;
```

atDrvrVersNum

This field returns the version number of AppleTalk if it is loaded.

sysVRefNum

This field returns the working directory reference number (or volume reference number) of the directory that contains the currently open System file. Use this reference number to read from or write to files in the System Folder.

HyperCard's Heap

Don't be sloppy with HyperCard's heap. As a rule you should be careful on the Macintosh because of the complexity of the Memory Manager. Dispose of any blocks you allocate and unlock any that weren't locked when you were called. Be careful with Toolbox routines that allocate memory. For example, if you display a dialog box using `GetNewDialog`, be sure to close it with `DisposDialog` rather than `CloseDialog`. A good rule of thumb during development is to check the Memory Manager's `FreeMem` and `MaxBlock` functions or HyperTalk's `the heapspace` function when your XCMD-

XFCN starts execution and just before it returns. The numbers should be close, if not identical, if your XCMD-XFCN properly disposed of all the memory it allocated.

See the Memory Manager chapters in *Inside Macintosh* Volumes II and IV for more details.

File I/O

Don't blindly open and close files. If you are doing file I/O operations, use the File Manager's `PBGetFInfo` function to see whether a file is open before accessing it. For those with experience in Macintosh File Manager hacking, if `pb^.ioFRefNum <>0`, a file is already open. If it was already open, be nice and restore its previous state when you're done (and don't close it).

See the File Manager chapter in *Inside Macintosh* Volume IV for more details.

Using *Get/WaitNextEvent*

If your XCMD-XFCN interacts with the user through calls to `GetNextEvent` or `WaitNextEvent`, you should be prepared to let the user switch out of HyperCard when MultiFinder is running. An example of this technique is the `ResCopy` XCMD. `ResCopy` displays a draggable window and allows the user to switch out of HyperCard by clicking another application window or the MultiFinder icon in the Menu bar. The way MultiFinder operates is that your application (or your XCMD-XFCN) is suspended during a call to `Get...` or `WaitNextEvent`. You don't have to worry about HyperCard starting to execute as your XCMD-XFCN's code contains the current event loop in your layer. Your XCMD-XFCN also should always make sure that its window is in front. You can avoid a switch by using `ModalDialog` rather than `GetNextEvent` or by having a dialog window (type `dBoxProc`) in front of the card window.

See "Macintosh Technical Note 158: Frequently Asked MultiFinder Questions" and "180: MultiFinder Miscellanea" for more information.

XCMD-XFCN Variables

XCMD-XFCNs can't use the 68000's register A5, which points to the application's global variables; it belongs to HyperCard. If you follow the building conventions, your XCMD-XFCN can use local variables as psuedo-globals.

However, this can cause problems. For one, you can't assume that you can use your XCMD-XFCN's variables from a `filterProc` or completion routine; the stack frame is out of sorts at that point and your psuedo-globals are probably not available. Suggested workarounds in order of preference are use a HyperCard global container instead, use a resource or temporary file, or stuff a handle to the relevant data into the ApplScratch area in low memory.

String constants are straightforward in Pascal; most compilers handle them

with no problems at all. However, C has a problem dealing with them. The problem is that string constants are usually declared as static variables. Statics require register A5, which is, of course, not available. Take caution in developing XCMDs-XFCNs in C for this reason. Most of the C publishers have developed workarounds for this; contact your publisher for more information.

The *XCmdBlock*

When an XCMD-XFCN is called by HyperCard, it is passed a data structure called the `XCmdBlock`. This block supplies the XCMD with all the relevant information it needs to perform its duties, including the arguments that HyperTalk is passing to it and the information needed for the callback mechanism to operate.

The fields are described in detail in the following text.

```
XCmdPtr = ^XCmdBlock;
XCmdBlock =
 RECORD
    paramCount  : INTEGER;
    params      : ARRAY[1..16] OF Handle;
    returnValue : Handle;
    passFlag    : BOOLEAN;
{ the fields above are directly accessed...}
{ the fields below are for HyperCard's use }
    entryPoint  : ProcPtr;
    request     : INTEGER;
    result      : INTEGER;
    inArgs      : ARRAY[1..8] OF LongInt;
    outArgs     : ARRAY[1..4] OF LongInt;
 END;
```

paramCount

Contains the number of arguments passed to the XCMD. This is available so that the XCMD-XFCN can cycle through the `XCmdPtr^.params` array below like this:

```
FOR i := 1 to XCmdPtr^.paramCount DO
  XCmdPtr^.params[i] . . . { do something here }
```

params

This is the array of `[1..paramCount]` (up to 16) handles to zero-terminated strings containing the arguments as passed from HyperCard.

returnValue

An XFCN uses this field to return its result. HyperCard disposes of this handle for you.

An XCMD can use this field to return the result.

passFlag

Setting this field to TRUE has the same effect as putting

```
pass messageName
```

into a HyperTalk script. For example, if your XCMD-XFCN is a replacement for a built-in HyperTalk command, you can pass control to the original HyperTalk command after your code is finished executing if you set `XCmdPtr^.passFlag := TRUE`. `passFlag is set to FALSE` by default.

entryPoint

For callbacks to work, HyperCard allows XCMD-XFCNs to actually jump back into the code of HyperCard itself. HyperTalk passes the address for callbacks in this field. (This is for internal use only.)

request

Contains a code corresponding to which callback is being made. (This is for internal use only.)

result

This should be examined every time you make a callback that could possibly result in an error, such as executing a script or another XCMD-XFCN. The value of this field is one of the following constants as defined in `HyperXcmd.p`.

```
xresSucc    = 0; { success }
xresFail    = 1; { failure }
xresNotImp  = 2; { not a valid HyperTalk expression }
```

In HyperCard 1.2 and beyond, XCMD-XFCNs no longer generate "Can't understand..." alerts; this field is the only indicator of whether a callback actually succeeded.

inArgs

Arguments as passed to callbacks. (This is for internal use only.)

outArgs

Arguments as returned from callbacks. (This is for internal use only.)

The *ChangeCase* XFCN Example

The following section decribes the `ChangeCase` XFCN example.

Description of ChangeCase

`ChangeCase` changes the capitalization of the characters in the container that it is passed and then returns the updated container as the function result.

`ChangeCase` begins at line 1900 with the BEGIN label. The logic flows as described in the flow chart shown in Figure 20-3. There are four decision points in the code.

First, on line 1980, `ChangeCase` determines whether the current version of HyperCard is greater than 1.1. If so, it sets the cursor to `busy` (this feature was not supported before version 1.2).

Next, on line 2020, `ChangeCase` determines whether the user passed the correct number of arguments (two); if not, the program terminates immediately.

Next, on line 2150, `ChangeCase` determines which of its procedures it should call depending on the mode specified by the user. If the mode is `kUpperMode` or `kLowerMode`, `DoSimpleCase` is called; if the mode is `kSentenceMode` or `kWordMode`, `DoComplexCase` is called.

Both `DoSimpleCase` and `DoComplexCase` determine the exact mode and operate on the data accordingly.

In `DoSimpleCase`, if the mode is `kUpperMode`, characters from `'a'` to `'z'` are forced to uppercase. If the mode is `kLowerMode`, characters from `'A'` to `'Z'` are forced to lowercase.

In `DoComplexCase`, if the mode is `kSentenceMode`, all characters from `'a'` to `'z'` that appear at the beginning of a sentence are forced to uppercase. If the mode is `kWordMode`. all characters from `'a'` to `'z'` that appear at the beginning of a word are forced to uppercase. In both of these cases, other characters are ignored.

ChangeCase *Syntax*

`ChangeCase` is passed two parameters and its syntax is

```
ChangeCase(container, mode)
```

`container` is any valid HyperCard container
`mode` is one of the following numbers:

```
1 { kUpperMode:   convert to uppercase }
2 { kLowerMode:   convert to lowercase }
3 { kWordMode :   capitalize words }
4 { kSentenceMode: capitalize sentences }
```

The following is an example using `ChangeCase` in a script

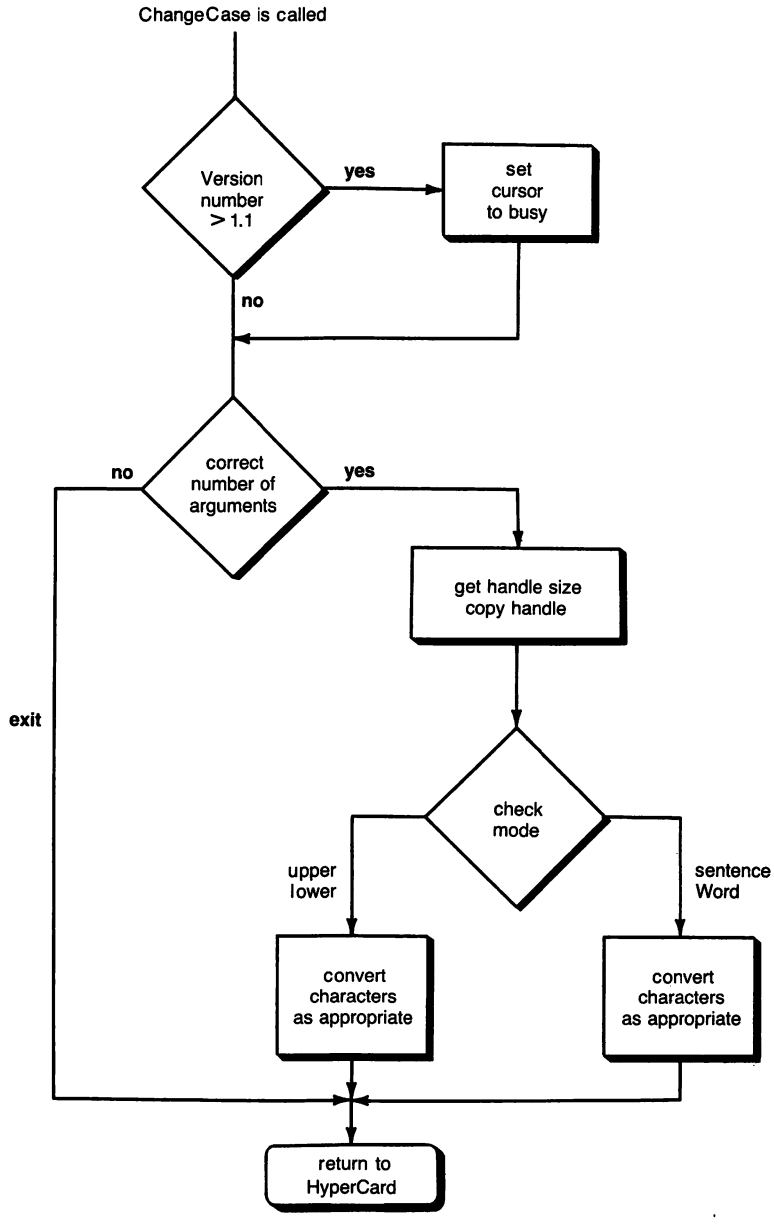


Figure 20-3. `ChangeCase` flow chart.

```
put ChangeCase("the quick brown fox",4) into the message box
```

The result is `The Quick Brown Fox`.

ChangeCase *Limitations*

`ChangeCase` has some limitations:

1. It can only accept containers of up to 32,767 characters because of Pascal limitations.
2. It sometimes doesn't handle diacritical marks properly. See the example of the word `Mañana` in the source code.
3. It is neither elegant nor terribly intelligent in its capitalization "algorithm," although it is a good general-case solution. This is just a nice way of saying "if it works, don't fix it."

The objective in developing `ChangeCase` is not to produce an ultimately powerful tool; rather, `ChangeCase` is meant as an example of XCMD-XFCN coding style and implementation tips.

Significant parts of the code are discussed in the text, and the source code comments are sufficient for the rest.

ChangeCase *Notes*

`ChangeCase` is written using MPW Pascal 2.0. (It is compatible with MPW Pascal 3.0 as well.) The building instructions are dependent on the following files all being in the same folder with the `ChangeCase` source code:

ChangeCase.p—ChangeCase's Pascal source code
ChangeCase XFCN—ChangeCase's HyperCard stack
HyperXCmd.p—Pascal interface file for the callbacks
XCmdGlue.inc—Pascal source code to the callback routines

The HyperTalk scripts used in this stack are included at the end of this chapter.

ChangeCase.p is broken down into relevant sections and each is described in detail.

The `ChangeCase` title card is shown in Figure 20-4.

ChangeCase Source Code

Each of the lines of code is referenced by its number in the complete listing.

Variables and Other Declarations

Here are explanations of variables and other declarations used in the program:

1-400 These are comments to help in development. Comments are one of the most effective tools a programmer can use. This format is widely used for small code examples.

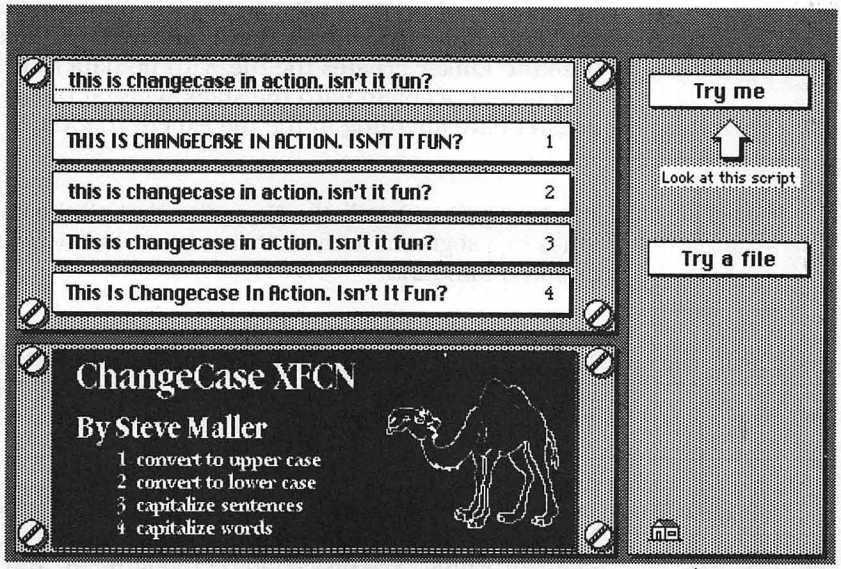


Figure 20-4. `ChangeCase` XFCN title card.

190-260 These are the MPW building commands. Highlight these lines and press the Enter key to build the XFCN.

200 Call the Linker. `-m ENTRYPOINT` tells the Linker to place the procedure called EntryPoint at the very beginning of the code. This is required for an XCMD-XFCN. `-rt XFCN=1999` tells the Linker to make an XFCN resource (rather than CODE) with the resource ID 1999.

Note: φ (Option-d) informs MPW that the command is continued on the next line.

210 `-sn Main=ChangeCase` tells the Linker that the resource should be named `ChangeCase` rather than the default name, `Main`.

220-240 `ChangeCase.p.o` is the output file from the compiler. This and the Macintosh Toolbox glue (`Interface.o`) and Pascal glue (`PasLib.o`) are linked together. Additional functionality, such as SANE, may require you to use more than just these object files.

250 `-o ChangeCase XFCN` tells the Linker the name of the file for output. In this case, it is the stack that you should have already created using HyperCard. Make sure that you have done this because otherwise the Linker creates the file with no data in it, and HyperCard may have a problem with the stack when it opens it because it doesn't have anything in the file except the XFCN.

440 `{$S ChangeCase }` tells the Compiler that all the code to follow belongs in a segment named `ChangeCase`. This works with the Linker option on line 210.

560-630 The `USES` declaration tells the compiler where the interfaces, or definition files, are for the rest of the source code. This is similar to the `#include` directive in C.

910-1050 Because an XCMD-XFCN can't have any global variables, we use the local variables of the `ChangeCase` procedure as globals. In this code, global variables are preceded by the letter `g`, and declared constants are preceded by the letter `k`. This is just a stylistic variation, but it leads to better readability.

1090 The XCmdGlue.inc file is included here. It is just as if the file was cut and pasted into this place in the source code. If you don't use any callbacks, you don't need to include this line in your source. This is similar to the `#include` directive in C.

ChangeCase (Main)

The main (outer loop) code of `ChangeCase` is a good example of a generic structure for an XCMD-XFCN. Here are descriptions of the significant lines in the code starting at line 1910.

1910 This is a trick to get the Pascal compiler to embed a copyright notice into the actual code. If you look at the XFCN resource using ResEdit, you see the string embedded in the code!

1980-2000 If the version of HyperCard is 1.2 or greater, we use a callback that sets the cursor to the beachball. If `ChangeCase` is run a number of times, the beachball spins because each call to `set cursor to busy` rotates the beachball.

2020-2030 These lines verify that the user actually passed two arguments. If

this is not the case, MPW Pascal's `EXIT` statement allows a graceful retreat. You should ideally alert the user to the situation with an error message using the answer dialog box. (Other Pascals may not support the EXIT statement; use a GOTO or a IF...THEN instead.)

2050 `CharsHandle` (part of TextEdit) is a data type that is defined as a Handle to a PACKED ARRAY [0..32000] OF CHAR (*Inside Macintosh* Volume I: page 384). This suits us just fine, although we are limited then to 32KB of text. No sweat; we can fix this in `ChangeCase` version 2.0!

2060 `GetHandleSize` is a Macintosh Memory Manager function that returns the size in bytes of the block to which the handle points. It is used here as an alternative to using a callback (for example: `the number of chars in...`).

2080 `paramPtr^.returnValue` is set to the handle of the text we're going to work on. This block normally has to be disposed of, but because we're going to reuse it, we just give it back and let HyperCard deal with it.

2100 `ZeroToPas` is a callback that converts a zero-terminated string to a Pascal string. `ChangeCase`'s second argument is the mode of operation; it should be 1, 2, 3, or 4.

2110 `StringToNum` is a Toolbox call (Package Manager) that converts the mode argument from a string to an integer we can use with the CASE statement.

2180 `DoSimpleCase` is the ALL CAPS or all lowercase procedure. It distinguishes whether we want upper- or lowercase.

2210 `DoComplexCase` handles the word and sentence capitalization. In these cases, the characters within the words or sentences are unaffected.

2250 `DisposHandle` is a Macintosh Memory Manager procedure that disposes of a relocatable block by freeing that block's master pointer. Because HyperTalk created this Handle specifically for us, we should dispose of it. In reality, this block is only two bytes long, but even that two bytes adds up if we're called a large number of times.

DoSimpleCase Procedure

`DoSimpleCase` is the all CAPS or all lowercase procedure. Note the problem mentioned in the comments at the top of the procedure.

1240 Here we loop from 0 to (the size of the text − 1). The reason this is adjusted is that the CharsHandle data type is indexed to start at zero, not one, as regular Pascal arrays are. `ORD` is a Pascal function that converts any ordinal type (numbers, characters, etc.) to the appropriate data type. In this case, we need a 16-bit integer, so this is handled by the compiler.

1260 `charCode` is used to hold the ASCII value of the i'th character in the container (`gText^^[i]`). This is to facilitate easier comparisons later in the procedure.

1290-1310 We check to see whether the i'th character is between `'a'` and `'z'`. If it is, we subtract the hex value 20 (32 in decimal) from the ASCII value, which forces it to be uppercase. CHR undoes the effect of ORD, turning the data type back into a character.

1350-1370 This is similar to the preceding routine used to do the uppercase, but we add $20 to the uppercase value to force it to lowercase.

DoComplexCase Procedure

`DoComplexCase` handles the word and sentence capitalization. In these cases, the characters within the words or sentences are unaffected.

1560-1590 A Pascal SET has an arbitrary number of elements. In this case, they are characters. SETs make complex comparisons easier because your code can check for membership with just one call rather than lengthy comparisons. This set defines the characters that break words or sentences.

1620 Just like in `DoSimpleCase`, we loop through the array of characters.

1650-1670 If the last character was a word/sentence break and this character is already capitalized, we're back inside a word/sentence, so we can turn off the `wasBreak` flag.

1690-1700 Set the `validChar` flag if this character is `'a'` through `'z'`; otherwise, clear it.

1720 We're going to capitalize the next word/sentence if the `wasBreak` flag is set and the next character is `'a'` through `'z'`.

1770 Now, we can turn off the `wasBreak` flag.

1800-1820 If we're in the middle of a word/sentence, we can check whether the next character is a word/sentence break. The potential problem here is that we can check for a break in the middle of a break and that's why line 1800 was put in. In other words, if `ChangeCase` is doing sentences and hits a space after a period, it tries to see whether that is a valid break, find out it isn't, and would not search out the start of the sentence one or two spaces away. Finally, we set `wasBreak` if the character is one of the SET for the appropriate mode.

Installation

In order to install an XCMD-XFCN, copy it into the resource fork of the stack from which it is to be used. You can use Apple's resource copying tool, ResEdit. Unfortunately, ResEdit requires you to quit HyperCard or be running MultiFinder with enough installed RAM to run both HyperCard and ResEdit (at least two megabytes). A better alternative is to use an XCMD called `ResCopy` (see Figure 20-5). It looks and works like Font/DA Mover with two notable advantages to the HyperCard user: it copies all resource types and it runs inside of HyperCard.

HyperTalk's inheritance rules affect an XCMD-XFCN just like a script installed in a stack: An XCMD-XFCN can only be called from that stack. If you install an XCMD-XFCN in your Home stack, it can then be accessed from any stack on your disk drive.

What Can You Do?

The question you might be asking at this point is: What can I as a talented and resourceful Macintosh developer do to enhance HyperCard?

The answer? It is entirely up to your creativity.

When HyperCard first arrived, there was no way to use the standard `open` and `save as` (Standard File package) dialogs from HyperTalk, so the author of this chapter wrote two XFCNs called `FileName` and `NewFileName` that fixed this limitation.

Following are a few ideas for XCMDs-XFCNs. This is by no means a closed list; let your imagination and ingenuity be your guide!

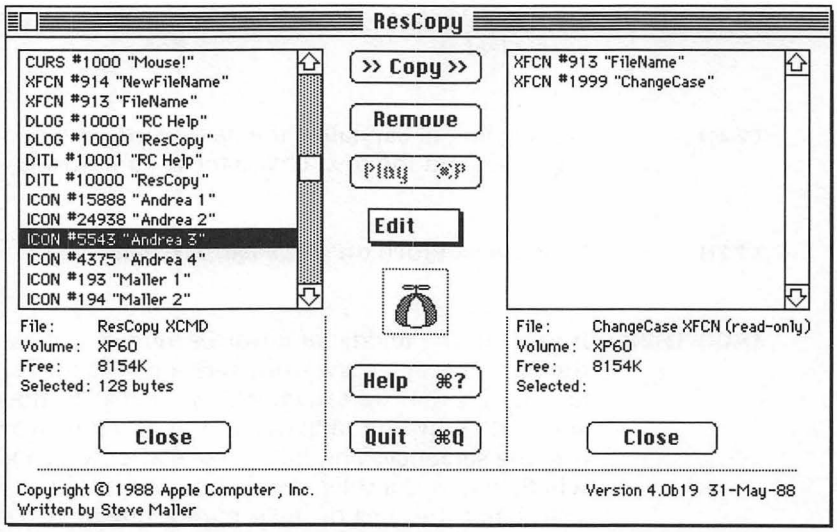


Figure 20-5. `ResCopy` XCMD.

Color Picture Display

This requires drawing directly onto the card window or opening a separate window. There are advantages to each approach. The advantage of drawing onto the card window is that the XCMD-XFCN is not modal, but the first time HyperCard redraws the screen, the graphic is lost. The advantage of drawing into another window is that HyperCard won't mess with that window. However, your XCMD-XFCN has to be modal in order to drag the window, for example.

Graphic Special Effects

For example, HyperCard could use the capability of doing 1° rotations, optical character recognition (OCR), etc.

Sound Decompression

You could write an XCMD, for example, which could decompress a compressed sound file and play it or that could even pass the message to the `play` command and get it to play the sound.

Access to Nontext Files

What about being able to read from a MacWrite or Word file or access records within a 4th Dimension database file or cells within an Excel spreadsheet? The `read from file` command is a good target for improvements.

ChangeCase.p Source Code

```
{[u+,r+,j=20-/55/,t=2,o=95,rec+,u+] PasMat formatting options}

{[f-] Turn MPW's Pasmat formatting off (cosmetic only) }

(*
ChangeCase: A HyperCard "XFCN" (External Function) Resource
            Version 1.0

Written By: Steve Maller for the HyperTalk Bible
            Copyright  1988 The Waite Group

Revisions:  stm  Tuesday, April 19, 1988    first pass; still missing modes 3 & 4
            stm  Wednesday, June 15, 1988   added new modes
                                              added callback for 1.2
                                              added comments

Language:   MPW Pascal

To build:
            Pascal ChangeCase.p
            Link  -m ENTRYPOINT -rt XFCN=1999
                  -sn Main=ChangeCase
                  -t STAK -c WILD
                  ChangeCase.p.o
                  {Libraries}Interface.o
                  {PLibraries}Paslib.o
                  -o "ChangeCase XFCN"

Usage:      ChangeCase(text, op)    -- text is a container or raw text to be operated upon
                                    -- op is 1, 2, 3, or 4:
                                        1 -- convert to upper case
                                        2 -- convert to lower case
                                        3 -- capitalize words
                                        4 -- capitalize sentences

Examples:   ChangeCase("hello, world.",1) -- returns HELLO, WORLD.
            ChangeCase("hello, world.",2) -- returns hello, world.
            ChangeCase("hello, world.",3) -- returns Hello, World.
            ChangeCase("hello, world.",4) -- returns Hello, world.

*)

{[f+] Turn MPW's Pasmat formatting back on (cosmetic only) }

{$S ChangeCase }                                    { This tells the MPW Linker to name the
                                                     resulting code (XFCN) resource
appropriately. If you skip this
directive, your XFCN will be named
"Main". }

UNIT Just_Say_No;                                   { this name is irrelevant }
```

continued

```
510
520   { =----------------------INTERFACE----------------------= }
530
540   INTERFACE
550
560     USES
570       {$LOAD PasSymDump}                          { use a packed MPW Pascal symbol file }
580       MemTypes,                                   { Memory Manager interface }
590       QuickDraw,                                  { QuickDraw interface }
600       OSIntf,                                     { Operating System interface }
610       ToolIntf,                                   { Toolbox interface }
620       PackIntf,                                   { Package Manager interface }
630       HyperXCmd;                                  { HyperCard's XCMD interface }
640
650     PROCEDURE EntryPoint(paramPtr: XCmdPtr);      { public declaration so that our routine
660                                                    can have an entry point, or a place
670                                                    where execution begins }
680
690     { =----------------------IMPLEMENTATION----------------------= }
700
710   IMPLEMENTATION
720
730     {$R-}                                         { turn off range checking }
740
750     TYPE
760       Str31              = String[31];            { used in the file "XCmdGlue.inc" }
770
780     PROCEDURE ChangeCase(paramPtr: XCmdPtr);      { declared forward so we can call it }
790       FORWARD;
800
810     { =----------------------EntryPoint----------------------= }
820
830     PROCEDURE EntryPoint(paramPtr: XCmdPtr);
840
850       BEGIN
860         ChangeCase(paramPtr);                     { call the XCMD code }
870       END;
880
890     { =----------------------ChangeCase----------------------= }
900
910     PROCEDURE ChangeCase(paramPtr: XCmdPtr);
920
930       CONST
940         kUpperMode         = 1;                   { convert to upper case }
950         kLowerMode         = 2;                   { convert to lower case }
960         kWordMode          = 3;                   { capitalize words }
970         kSentenceMode      = 4;                   { capitalize sentences }
980
990       VAR
1000        gTempStr          : Str255;               { scratch string }
1010        gTempLong         : LongInt;              { scratch 4-byte integer value }
1020        gMode             : Integer;              { the four modes which we accept }
1030        gTheChar          : Char;                 { scratch character }
```

continued

```
1040        gText              : CharsHandle;            { see IM Volume 1: page 384 }
1050        gTextSize          : LongInt;                { In theory this can be huge, but in
1060                                                      practice, Pascal doesn't like large
1070                                                      arrays, so we have to be careful... }
1080
1090        {$I XCmdGlue.inc }                           { now compile the XCMD callback code }
1100
1110        { =----------------------DoSimpleCase----------------------= }
1120
1130      PROCEDURE DoSimpleCase;
1140
1150  { Note: we don't correctly handle diacritical marks. For example,
1160  type this in the message box: put ChangeCase("Mañana",1) into msg
1170  This *should* put "MANANA" into the msg, but instead you get MAñANA! }
1180
1190        VAR
1200          i                  : Integer;              { counter variable }
1210          charCode           : Integer;              { ASCII code of the i'th character }
1220
1230        BEGIN
1240          FOR i := 0 TO ORD(gTextSize - 1) DO        { Convert to an INTEGER value }
1250            BEGIN
1260              charCode := ORD(gText^^[i]); { Check the 'i'th character }
1270              IF gMode = kUpperMode THEN             { Are we doing ALL CAPS? }
1280                BEGIN
1290                  IF (charCode >= $61) AND           { If we're >= "a"... }
1300                     (charCode <= $7A) THEN          { ...and <= "z" }
1310                    gText^^[i] := CHR(charCode - $20); { make it UPPER CASE }
1320                END
1330              ELSE                                   { We are doing lower case }
1340                BEGIN
1350                  IF (charCode >= $41) AND           { If we're >= "A"... }
1360                     (charCode <= $5A) THEN          { ...and <= "Z" }
1370                    gText^^[i] := CHR(charCode + $20); { Make it lower case }
1380                END;                                 { IF gMode = kUpperMode THEN }
1390            END;                                     { FOR i := 0 TO ORD(gTextSize - 1) DO }
1400        END;                                         { DoSimpleCase }
1410
1420      { =----------------------DoComplexCase----------------------= }
1430
1440      PROCEDURE DoComplexCase;
1450
1460      VAR
1470        i                  : Integer;              { counter }
1480        charCode           : Integer;              { ASCII code of the i'th character }
1490        wasBreak           : Boolean;              { did we just break a word? }
1500        validChar          : Boolean;              { do we want to process this one? }
1510        breakChar          : SET OF Char;          { WORD break characters }
1520
1530      BEGIN
1540
1550        IF gMode = kSentenceMode THEN
1560          breakChar := [CHR(13), '.', '?', { These are the valid breaks for }
```

continued

```
                    '!', ''] { SENTENCES... }
        ELSE
          breakChar := [CHR(13), ' ']; { These are the only ones for WORDS }

        wasBreak := TRUE;                          { so we get the first character!}
        FOR i := 0 TO ORD(gTextSize - 1) DO        { loop through all the chars }
          BEGIN
            charCode := ORD(gText^^[i]); { copy the 'i'th character }
            IF wasBreak AND (charCode >= $41) AND  { If we're >= "A"... }
               (charCode <= $5A) THEN              { ...and <= "Z" }
              wasBreak := FALSE;                   { flag: we're still in a word }

            validChar := (charCode >= $61) AND     { If we're >= "a"... }
                         (charCode <= $7A);        { ...and <= "z" }

            IF (wasBreak) AND (validChar) THEN
            { if we're in a word/sentence break and the current char is a real char }
              BEGIN
                gText^^[i] := CHR(charCode - $20); { then force uppercase }

                wasBreak := FALSE;                 { flag: we're still in a word }
              END;                                 { IF wasBreak... THEN }

            IF NOT wasBreak THEN
              IF (CHR(charCode) IN breakChar) THEN
                wasBreak := TRUE;                  { set flag if we're in a word break }

          END;                                     { FOR i := 0 TO ORD(gTextSize - 1) DO }
      END;                                         { DoComplexCase }


      { =---------------------- * ChangeCase * ----------------------= }

      BEGIN                                          { PROCEDURE ChangeCase }
        gMode := StringWidth('ChangeCase 1.0  1988 The Waite Group');

{ First, we use a callback to determine if the current version of HyperCard
supports the beachball cursor. This use of EvalExpr will return 'true' if
the expression evaluates true. This means that we are running at least
version 1.2... }

        ZeroToPas(EvalExpr('the version > 1.1')^, gTempStr);
        IF gTempStr = 'true' THEN
          SendCardMessage('set cursor to busy');      { put up the beachball cursor }

        IF paramPtr^.paramCount <> 2 THEN { Oops, wrong # of arguments! }
          EXIT(ChangeCase);                           { Tell MPW to jump to the end }

        gText := CharsHandle(paramPtr^.params[1]);    { convert the handle to suitable type }
        gTextSize := GetHandleSize(Handle(gText));    { see how much is in it }

        paramPtr^.returnValue := paramPtr^.params[1]; { copy the handle to returnValue }
```

continued

```
2100        ZeroToPas(paramPtr^.params[2]^, gTempStr); { get the mode - it is a string here }
2110        StringToNum(gTempStr, gTempLong);            { StringToNum (Package Manager) returns a
2120                                                      32 bit number }
2130        gMode := ORD(gTempLong);                     { scale gTempLong to a 16-bit integer }
2140
2150        CASE gMode OF                                { What exactly are we doing here }
2160
2170          kUpperMode, kLowerMode:                    { We're operating on all characters }
2180            DoSimpleCase;
2190
2200          kSentenceMode, kWordMode:                  { We're doing words or sentences only }
2210            DoComplexCase;
2220
2230        END;                                         { CASE gMode OF }
2240
2250        DisposHandle(paramPtr^.params[2]); { We're responsible to dispose of this }
2260
2270      END;                                           { PROCEDURE ChangeCase }
2280
2290    END.
```

Appendix A

HyperCard-HyperTalk Version History

The following describes features, enhancements, and bug fixes to HyperCard that are related specifically or indirectly to the HyperTalk language. This includes things that would be of interest to anyone developing stacks with HyperCard. Previously, HyperCard was released as Version 1.0.1 and then 1.1. This description starts with Version 1.2, which was the first major upgrade that affected scripting, and continues to Version 1.2.2.

Note that all versions of HyperCard are backwardly compatible. Thus, scripts written for a specific version of HyperCard are only guaranteed to work on the same or earlier versions of HyperCard. The value returned by the `version` function (a 1.1 feature that was enhanced in 1.2) can be used to ensure that the user has the correct version of HyperCard. (See chapter 14, "Advanced Commands and Functions.")

Features Added to HyperCard Version 1.2 and Related to HyperTalk

Features and improvements to Version 1.2 of HyperCard are mainly two new system messages for handling the Return and Enter keys in fields and three new HyperTalk commands (`lock screen`, `unlock screen`, and `select`). The `find` command has two new options, and the `hide` and `show` commands now work on background and card pictures (paint layers). There are two enhancements to existing functions and nine new functions. These cover finding the `version` of HyperCard being used, dealing with found text returned with the `find` command, dealing with selected text, and a `screenRect()` function. Five new properties have been added to HyperCard 1.2, including properties for dealing with CD ROMs and read-only stacks (`cantModify` and `cantDelete`) and eight new cursor names. There are eight new ways to refer to parts of screen rectangles, buttons, and fields. Version 1.2 also adds 12 new synonyms (abbreviations) for HyperTalk terms. There are additional keyboard shortcuts for opening the scripts of objects. These new features have been covered in the various chapters of this text. See the index for further information.

Features Added to HyperCard Version 1.2.2 and Related to HyperTalk

Features and improvements to Version 1.2.2 of HyperCard are mainly an upgrade for improved support of displaying and printing Japanese, Arabic, and other non-Roman languages. Also, there are improvements in the sound manager for working with System 6.0.2, improved robustness in low-memory situations, parameters returned by the `find` command are enriched, pasting of fields and text is improved, and a variety of bugs are fixed. The following sections of this appendix describe these 1.2.2 enhancements. They are not covered in the chapters of this book. Note that HyperCard Version 1.2.1 was a minor bug fix for HyperCard and had no significant feature improvements.

Enhancements to HyperCard 1.2.2 Related to Scripts

Here are the 1.2.2 scripting-related enhancements.

Pasting Fields with the Shift Key Down

In previous versions of HyperCard, when you pasted a field, only the field itself was pasted; the text in the field was lost. Now, holding down the shift key when you paste a field causes the text in that field to also be pasted. A script can simulate pasting a field and its text as follows.

```
on mouseUp
   select cd fld 1
   doMenu "Copy Field"
   go last card
   type "V" with commandKey, shiftKey -- simulates Command-V paste with
-- shift key
   choose browse tool
end mouseUp
```

Placing Paint Graphic While Drawing

It is possible in HyperCard 1.2.2 to draw a rectangle, oval, or regular polygon, and then place it while it is being created (placing means sliding it around the screen). If the command key is pressed with the mouse still down while the shape is dragged, the stretching of the shape stops and the graphic may be slid around the screen for placement. Releasing the command key allows stretching to continue.

Faster Navigation

Going to the previous card of a specific background is considerably speeded.

Bug Fixes Related to Scripts

Here are some of the major bug fixes to HyperCard 1.2.2.

Low Memory and Sound

Sometimes, when running HyperCard on a MacPlus under low-memory conditions, when you repeatedly copied and pasted an object, the stack could become corrupted and cause the system to hang. This is fixed.

The Macintosh 6.0.2 fixed a number of problems related to the sound manager. There were also problems with HyperCard 1.2 and the system sound manager, and this is supposedly fixed in the new 1.2.2.

Find Bugs Fixed

The `find` command is intended to be a very powerful way to locate information in HyperCard. One of the ways that you can use `find` is to intercept it periodically from inside a handler, check its results thus far, and modify, terminate, or override it. To do this, you use `the params` to read information about the search. For example, if you used the `find` command:

```
find whole "Big Person" in bkgnd fld "People"
```

and you intercepted it to examine `the params`, you find the following literal results in Version 1.2.1 and earlier:

```
find whole Big Person in <field>
```

In Version 1.2.2, HyperCard now returns:

```
find whole "Big Person" is bkgnd fld 2
```

This is better, because it is easier to identify the name of the field as being item three of the list. In 1.2.1 and earlier, you had to figure out that the name was broken into two parts, which was not easy. Note that HyperCard converts field names used in the `find` command to field numbers when they are read as parameters.

Choose Bug Fixed

In pre-1.2.2 versions of HyperCard, when you clicked a tool, HyperCard did not properly send the `choose` message. This is fixed and is very useful when you want

your script to monitor and figure out which tool was clicked. Note that the `choose` message passes the word `"tool"` and a number, not the tool name.

```
on choose what, whichTool
   if what is "tool" and whichTool is "7" then beep
   pass choose
end choose
```

Copying the selection

In pre-1.2.2 HyperCard, if a handler requested that `the selection` (the highlighted text) be put after the contents of a destination, it accidentally copied the entire text of the source container. This is fixed.

diskSpace( )

The `diskSpace()` function now returns the correct values for disk space when it is greater than 32 megabytes.

Answers to Review Questions

Chapter 1:

1. The correct answer is b. (The book described by c is *The Waite Group's Tricks of the HyperTalk Masters*, published in 1989 by Howard W. Sams & Company.)
2. a, b, c, e. The answer d is not available at the time of this writing.
3. d. The answer e is not a legal user level.
4. b
5. b. All will work, but b is the best way because you will be able to move from Home to the HyperTalk Bible stack without moving into another folder. Answer c is also useful if you want to add lots of stacks to your HyperTalk Bible and keep them separate from the standard stacks found in the HyperCard Stacks folder where Apple puts the Home stack.
6. True
7. False
8. It beeps three times.
9. b
10. True
11. a
12. c
13. False. They can be in cards, backgrounds, stack, or the Home stack.
14. e
15. False. It is used precisely for trying out commands.
16. Yes. This is useful on large screen Macs where the OK button to close the editor is at the bottom of the screen.
17. c
18. d. There is nothing to `put`. By the way, answer c works even though `beep` is a keyword.
19. b
20. It plays the boing sound and puts the text `"...heres the Message box"` into the Message box.
21. a
22. False

23. c
24. It moves the upper-left corner of the Message box to the upper left corner of the card.

Chapter 2:

1. False. The HyperCard card is also missing a resize and go-away box.
2. d
3. c
4. a
5. a
6. c
7. False. A button on a background can be seen by all cards that have it as a background.
8. d
9. b
10. b
11. d
12. False. It's dangerous to change the user's Home stack. It is safe to change your own if you know what you are doing.
13. a
14. b
15. False
16. c
17. a and c
18. a
19. False. You can show and hide buttons, fields, card graphics, or background graphics.
20. a or b works
21. b
22. c
23. a
24. False. Pathnames are the names of the folders and disks that contain a stack.
25. b
26. using `push` and `pop`
27. e and f are illegal

Chapter 3:

1. tempo
2. boing, harpsichord, and silence
3. by letter (A–G) or by number (frequencies)
4. no voice, H is not a valid note, and the tempo is out of place
5. `play "harpsichord" tempo 150 "f g"`
6. nothing

7. `fast` is out of order, should be `venetian blinds`, no `close` with `venetian blinds`, and `next card` is not a valid image
8. `choose field tool`
9. `doMenu "Bkgnd Info..."`
10. `ask` and `answer`
11. `answer "Erase hard disk?" with "NO!!!" or "I guess"`
12. `ask "This computer is a..." with "Macintosh"`

Chapter 4:

1. a field
2. the Message box
3. variable
4. `the selection`
5. **a.** invalid, contains a space, **b.** valid, **c.** valid, **d.** invalid, starts with a number, **e.** invalid, contains a period
6. local
7. **a.** h, **b.** f, **c.** g, **d.** h
8. **a.** g, **b.** e, **c.** h, **d.** f

Chapter 5:

1. **a.** p, **b.** l, **c.** n, **d.** o, **e.** i, **f.** k, **g.** j, **h.** m
2. **a.** `note^2 + base`, **b.** `trunc (base * note^pi)`, **c.** `subtract (round(note/three)) from card field foo`, **d.** `average(round(pi), round(pi), round(pi))`
3. Yes
4. **a.** 8, **b.** ? (overflows 72-digit limit), **c.** a random number between 0 and 10000000000, **d.** the remainder of dividing the numbers 1 to 6 by pi (For numbers 1, 2, and 3, the result is 1, 2, and 3. For numbers 4, 5, and 6, the number is a decimal fraction.)
5. Put the following script into a button:

```
on mouseUp
  put 123456.7890 into x
  set the numberFormat to "0.00"
  put "$" & x into msg
end mouseUp
```

6. No, you can't get the global property `numberFormat` to take effect when you enter it from the Message box. You must place it in a handler. When the handler completes (`on mouseUp finished`) HyperCard goes to the idle mode, and `numberformat` returns to its default situation. As long as HyperTalk is executing any script, `numberFormat` stays in effect.
7. Surprising results: **a.** 2, **b.** 2 (Although most books tell you the rule is to round up when a number is .5 or greater and round down when it's less than .5, this is

not how the algorithm is implemented in HyperTalk (SANE). Remember, in HyperTalk, round down if the number is even and up if it's odd.)

8. This handler uses the `mod` operator to make sure that as `note` grows, it is reset to 0 when it hits 40 and starts over:

```
on mouseStillDown
  put (note + 1) mod 40 into note
end mouseStillDown
```

9. **a.** This statement is broken down by starting inside and working out. It generates three ranges of random numbers: 1 to 100, 1 to 200, and 1 to 300. The average of 100, 200, and 300 is 200. The result is the average of three random numbers that hovers in general around the 200 mark, but swings between 1 and 300 with ferocity.
   b. This result is the division of two random number generators, one running from 1 to 100 and the other from 1 to 5. The result is a range of even more random numbers running from 1/5 to 20, which are then rounded so that there are no decimals.

Chapter 6:

1. an argument or a parameter
2. a handler
3. `mouseEnter`
4. `idle`
5. `arrowKey`, `controlKey`, and `functionKey`
6. d
7. `mouseDown` and `mouseUp` with `mouseWithins` and `idles` before and after
8. `openField`
9. `closeCard`, `closeBackground`, `openBackground`, and `openCard`
10. `startUp`
11. `suspend`
12. `idle`
13. a
14. b
15. at the card
16. `send mouseUp to card button id 3`
17. a

Chapter 7:

1. Use `the mouseH` or item 1 of `the mouseLoc`.
2. `the mouseLoc` returns the mouse position at the moment the function is executed, and `the clickLoc` returns the mouse position at the moment of the most recent mouse click.
3. No. `on mouseDown` detects the act of pressing the mouse button; it detects a mes-

sage. The mouse detects whether the mouse button currently is down; it reports on the continuing state of the mouse. It can't be used to initiate a message handler.

4. **a.** h, **b.** i, **c.** j, **d.** f, **e.** g
5. Yes. The second `the mouseClick` detects whether there had been any clicks since the first use of `the mouseClick`.
6. `click at the clickLoc with OptionKey`
7. Use an on mouseEnter handler in the card rather than the on mouseWithin handler.

Chapter 8:

1. The problem is that `optionKey` is a function, not a message, so it can't be used to initiate a message handler.
2. **a.** function, **b.** function, **c.** message, **d.** message, **e.** function, **f.** message
3. The `ask` command places the user's answer into the variable `it`. The `ask password` command first encrypts the answer into a number, and then places that number into `it`.
4. Nothing. The message never gets sent to the button. (However, you could use a `send` command on, say, the card level to send the `returnKey` message to a button.
5. **a.** left, **b.** 5, **c.** down
6. Keyboard messages aren't used in button scripts because these messages enter the HyperCard system at the card level, so they are never seen at the button level (unless, of course, you cheat and use a `send` command to send one to a button).
7. c

Chapter 9:

1. **a.** `sam is sally` or `sam = sally`, **b.** `sam <> sally`, **c.** `sally > sam` or `sam < sally`, **d.** `sam > sally` and `sam < 40`
2. **a.** should be a `then` before the `put`, **b.** should be either all on one line (perhaps continued with ¬) or in one of the following two forms:

```
if it is yes
   then put "You ask too much of me" into the message box


if it is yes then
     put "You ask too much of me" into the message box
end if
```

c. This is ok; the `else` acts to delimit the first `if`, **d.** The first line should end with `then`, and the last line should be followed by the line `end if`.

3. **a.** the mouseH = 200 and the mouseV = 300, **b.** the shiftKey is down or the optionKey is down, **c.** the mouseClick and the shiftKey is down, **d.**

```
(the shiftKey is down and the optionKey is up) or¬
(the shiftKey is up and the optionKey is down)
```

4. if the commandKey is down then beep
5.
```
if it is "Capital" then put "Austin" into the message box
   else if it is "State" then put "Texas" into the message box
   else exit mouseUp
```

   Because the ask command is limited to three choices, you don't need to test explicitly for the Cancel response.

Chapter 10:

1. **a.** 3, **b.** 7, **c.** 5, **d.** 1, **e.** 0, **f.** 10, **g.** 0 if the first guess is correct, an infinite number otherwise, because the answer command doesn't provide any way for the global variable it to be set to 8.
2. **a.** 3, **b.** 12, **c.** 24, **d.** 30, **e.** 54
3. repeat while, repeat until, wait while, wait until
4. The next repeat command causes the program to skip the rest of the loop and start the next cycle of the loop, but the exit repeat command causes the program to leave the loop entirely.
5. **a.** When the loop variable note reaches 70, one final cycle is executed and the loop terminates, **b.** The loop terminates when the user holds the Shift key down at the time the loop test is made. **c.** The loop terminates when the mouse is clicked or when the horizontal mouse location is greater than 500. Pressing the option key just skips one of the beeps in the loop.
6. It marks the end of the list of commands that are executed during each cycle of the loop.
7. **a.** No, a repeat with loop has no mechanism for testing relationships. **b.** Yes. For instance, the following two loops have the same beeping effect:

```
repeat with n = 1 to 8
  beep n
end repeat

put 1 into n
repeat while n <= 8
  beep n
  add 1 to n
end repeat
```

   However, the first form executes faster. Also, the first form leaves the variable n with the value 8 after the loop finishes; the second loop leaves n with the value 9.
8. Yes, yes, yes, and yes, as long as the values are integers.

Chapter 11:

1. The & operator glues two strings directly together, but the && operator inserts a space between the two.
2. "Soak the oak," she said.
3. a, c, e, and f (b should use card field, not just card; d should be line 4 to 6, not line 4 to line 6.)
4. a, c, and e (in f, `the selectedText` is a function, not a container.)
5. Here are three:

```
put the length of card field 1 into fieldCount
put length(card field 1) into fieldCount
put the number of characters in card field 1 into fieldCount
```

6. put `return & newstuff` after card field Big Things
7. By using the modifiers `chars in`, `words in`, `items in`, or `lines in`.
8.
```
if "beep" is in possibilities then beep
if possibilties contains "beep" then beep
if offset("beep", possibilities) > 0 then beep
```

9.
```
put character 1 of it into firstchar
if charToNum(firstchar) >= charToNum("A") and ¬
charToNum(firstchar) <= charToNum("Z") then beep
```

   You have to go to the ASCII values to distinguish capital letters from lowercase letters.

Chapter 12:

1. **a.** `the date` or `date()` or the `short date`, **b.** `the long date`, **c.** `the abbreviated date` or `the abbrev date` or `the abbr date`
2. No. The function is called once, just when the mouse button is pressed down. The date value returned at that moment is assigned as the new button name.
3. No. If the quotes are omitted, HyperTalk interprets the / symbol as the division operator, so 4/15/88 is evaluated to 0.0030303.
4. The Control Panel on your Mac (accessed via the Apple Icon on the Menu Bar) is set to the 12-hr clock; her's is set to the 24-hour clock.
5. **a.**
```
convert car to long date
put car into result
```

   b.
```
convert car to long date
put item one of car into result
```

   c.
```
convert car to seconds
put the seconds - car into result
```

6. `the seconds` returns the number of seconds since 00:00:00, Jan 1, 1904, but `the ticks/60` returns the number of seconds since the Mac was last restarted. Unless you've tinkered with the Mac's clock, these two times are not the same.

Also, in general, the ticks/60 has a decimal part, but the seconds always is an integer.

7.
```
put the long date into now
put "It is" && the time && "on a" && item 1 of now && "in"
&& word 1 of¬
item 2 of now & "." into the message box
```

8.
```
convert the long time to dateItems
put "The time is" && item 4 of it && "hours," && item 5 of it &&¬
"minutes, and" && item 6 of it && "seconds."
```

Chapter 13:

1. **a.** max(sales), **b.** average(sales), **c.** average(max(sales), min(sales))
2. a random number in the range 21-28
3. **a.** 1000*compound(0.10,15), **b.** 1000*compound(0.10/4, 4* 15) or 1000*compound(0.025,60), **c.** 1000*compound(0.10/12, 12*15) or 1000*compund(1/120, 180)
4. the annuity() function
5. **a.** e, **b.** f, **c.** d
6. A = atan(3/4) = 0.64 radians = 37 degrees (rounded)
7. **a.** yes, **b.** no, **c.** yes, **d.** no
8. **a.** sin(30 * pi / 180) or sin(pi/6), **b.** exp2(10), **c.** exp(1/3), **d.** log2(45)

Chapter 14:

1. **a.** big, **b.** 2 parameters, **c.** x and y, **d.** j, **e.** n
2. **a.** 112 is a very big number, **b.** word 1 of verybig(4,-8)
3. c
4. When HyperTalk encounters a pass thisMessage or exit thisMessage command in a handler, it terminates the handler. In the case of the exit command, that also marks the termination of the message. The pass command, however, sends the message on to following levels of the hierarchy.
5. **a.** normal field: No, because you can't click a normal field; the browse tool turns into the text tool automatically, **b.** locked text field. Yes, **c.** card. Yes, **d.** button. No, because buttons normally have their own mouseUp handlers. You can make it work for buttons by putting a pass mouseUp command into the button scripts.
6. The target() function is used in a message handler, and it returns the name of the object that first received the message currently being processed.
7. The result() function is used in conjunction with the go to or find command in a script. It returns empty if the command is successful and Not found otherwise.
8. You can use the on <command-name> <parameter-list> format. Or you can use the function the param of <number> to access particular parameters.
9. No. Just the parameter list approach works.

Chapter 15

1. b and d are legal forms
2. **a.** k, **b.** n, **c.** l, **d.** m, **e.** j, **f.** p, **g.** i, **h.** o
3. **a.** `play "harpsichord" "f3 g a b c4 d e f"`, **b.** `play "harpsichord" 75`, **c.** `play "boing" tempo 16000 "c6w c c c

8. with `doMenu "Select"`
9. with `doMenu "Select"` and `doMenu "Invert"`
10. with `doMenu "Transparent"`
11. no

Chapter 18:

1. **a.** `the blindtyping`, **b.** `the editBkgnd`, **c.** `the lockScreen`, **d.** `the location of the Message box` or `loc of msg` for short, **e.** `the userLevel`, **f.** `the userModify`
2. b, c, d, e
3. **a.** should be no `the` before `tool window`. (It's okay to omit `the` before `loc`.), **b.** should be a `the` before `userLevel`, **c.** use `busy` rather than `6`; HyperTalk only recognizes numeric labels for the first four cursor choices, **d.** should be `topLeft` (or `loc`), not `upperLeft`
4. none
5. The `cursor` property
6. the `rectangle`, `width`, and `height` window properties

Chapter 19:

1. y
2. a
3. delimiter
4. y
5. stack, folder, file
6. **a.** valid, **b.** valid, **c.** invalid
7. `write message box & tab to file "MessageFile"` (or `write message...` or `write msg...`)
8. in the variable `it`
9. b
10. a
11. b
12. You can't read after a write.

Answers to Exercises

Chapter 1:

Use the prototype handler we used in Example 11. Give the new button the name "myButton" and change the script in it to

```
on mouseUp
   set the loc of card button "myButton" to ¬
   random(100)+200,random(75)+100
end mouseUp
```

Note the use of the option-return character so that the line can wrap properly.

Chapter 2:

1. Enter in the card script

```
on mouseUp
  put "mouseUp in card" into msg
end mouseUp
```

2. Into the card script, put

```
on mouseUp
  put "mouseUp in card" into msg
  pass mouseUp
end mouseUp
```

Into the background script, put

```
on mouseUp
```

continued

```
  put "mouseUp in bkgnd" into msg
end mouseUp
```

3. Enter in a new button

```
on mouseUp
  put the long name of me into the msg box
end mouseUp
```

4. Enter in a new field

```
on mouseUp
  go any card
end mouseUp
```

Be sure the field is locked for this script to work.

Chapter 3:

1.
```
on mouseUp -- button handler
  answer "Play..." with "boing" or "harpsichord"
  play it tempo 400 "60 61 62 63 64 65 66"
end mouseUp
```

2.
```
on mouseDown -- card script handler
  visual effect checkerboard to inverse
  visual effect dissolve slow
  go to this card
end mousedown
```

3.
```
on mouseUp -- button handler
  answer "Select which tool?" with "field" or "button"
  play "boing" tempo 500
  choose it tool
end mouseUp
```

Chapter 4:

Make sure numeric values are in the variables before clicking the button.

```
on mouseUp -- Add button
  global var1,var2
  add var1 to var2
end mouseUp
```

Chapter 5:

1. Here is the script for the Random Music button and the two Tempo buttons:

```
on mouseStillDown -- button "Random Music"
  global note, rate
  put random(40) + 40 into note
  play "harpsichord" tempo rate note
end mouseStillDown

on mouseUp -- button "tempo 500"
  global rate
  put 500 into rate
end mouseUp

on mouseUp -- button "tempo 1000"
  global rate
  put 1000 into rate
end mouseUp
```

2. Here is the script for the Sweep button. It uses `mod` to keep the variable from overflowing.

```
on mouseStillDown -- button Sweep
  global note, rate
  put (note + 1) mod 60 into note
  play "harpsichord" tempo rate note + 20
end mouseStillDown
```

Chapter 6:

1.
```
on mouseUp -- Next button
  answer "Go to next card with" with "wipe left" or "scroll left"
  send music -- music is a custom subroutine/handler in the card
-- script
  visual effect it -- the variable it contains the visual effect
-- parameter
  go next
end mouseUp

on mouseUp -- Previous button
  answer "Go to previous card with" with "wipe right" or "scroll ¬
  right"
  send music
  visual effect it
  go to previous
end mouseUp

on music -- Messages card
  dial "10901"
end music
```

2.
```
on mouseStillDown -- Messages card
  answer "Where to boss?" with "Next" or "Previous"
  -- the preceding command places one of the two button names
  -- into the variable it. The next line sends a message to that
  -- button,
  -- which is the equivalent of clicking it. Note that one message
  -- goes
  -- down to the button, which sends another message up to the
  -- music
  -- handler in the card script.
  send mouseUp to card button it
end mouseStillDown
```

Chapter 7:

1. Here is one approach, which puts empty space after the click location to separate it from the mouse location.

```
on mouseUp
  put the clickLoc into the message box
  put " " after the message box
  put the mouseLoc after the message box
end mouseUp
```

After reading about strings in chapter 11, you probably would do it this way:

```
on mouseUp
  put the clickLoc && the mouseLoc into the message box
end mouseUp
```

2. In analogy to Example 12, we can do the following: (The main difference is that the on mouseWithin handler is in the button rather than in the script.)

```
on mouseUp
  put the mouseLoc into the message box
end mouseUp

on mouseWithin
  click at the mouseLoc
end mouseWithin
```

It's simpler to dispense with the `click at` command and just use the following:

```
on mouseWithin
  put the mouseLoc into the message box
end mouseWithin
```

Chapter 8:

1. Place these handlers into the button script.

```
on mouseEnter
  if the shiftKey is down then beep
end mouseEnter

on mouseLeave
  if the optionKey is down then play "harpsichord"
end mouseLeave
```

2. Place the following command into a button or into the Message box:

```
answer "What do you want to do now?" with "quit" or "repeat" or¬
"continue"
```

   Remember that the last choice is the default—the one that gets done if you press the Return or Enter key.

3. First, create a field called `Capitals`. Next, enter `ask password "Where?"` in the Message box. Reply with `Sacramento` in the ask box. Enter `put it into card field Capitals`. This places the encrypted answer into the field. Then, create a button and give it this script:

```
on mouseUp
  ask password "What's the capital of California?"
  if it is word 1 of card field "Capitals" then put "Right!"
  if it <> word 1 of card field "Capitals" then put "Wrong!"
end mouseUp
```

4. Remember to place the script into a card, background, or stack, not a button or field.

```
on arrowKey direction
  if direction is up then play "harpsichord" b
  if direction is down then play "harpsichord" c
  pass arrowKey
end arrowKey
```

Chapter 9:

1.
```
on mouseUp
  if the shiftKey is down then go to next card
end mouseUp
```

2.
```
on mouseUp
  if the shiftKey is down then go to next card
  else if the optionKey is down then go first card
end mouseUp
```

   One problem with this script is that if both keys are down, the script goes to

the next card because once it finds that the Shift key is down, it doesn't check the `else` part. The next script fixes that problem.

```
on mouseUp
  if the shiftKey is down and the optionKey is up then go to next
card
  else if the optionKey is down and the shiftKey is up
  then go first card
end mouseUp
```

3.
```
on openCard
  ask "What's your name?"
  if it is empty then exit openCard
  else put it into card field 1
end openCard
```

4. First, create a card field called `Desserts`. Then, put the following script into it:

```
on mouseEnter
  if card field "Desserts" is empty then
    ask "What's your favorite dessert?"
    put "My favorite dessert is " into card field "Desserts"
    if it is empty then put "Chocolate-dipped pickles" into it
    put it after card field "Desserts"
  else
    ask "Name another dessert you like:"
    if it is empty then exit mouseEnter
    else
      put " I also like " after card field "Desserts"
      put it after card field "Desserts"
    end if
  end if
end mouseEnter
```

In the initial `if then` segment, the global variable `it` is set to the user's input. If the user fails to provide input (i.e., if `it` is `empty`), the script provides a default value. Chapter 11, "Strings," provides alternative ways for combining `it` with a preexisting string, such as `I also like`.

Chapter 10:

1. Here's one solution.

```
on mouseUp
  ask "What number is the lower limit?"
  if it is empty then put 0 into lower
  else put it into lower
  ask "What number is the upper limit?"
  if it is empty then put 10 + lower into upper
  else put it into upper
  put 0 into sum
```

continued

```
  repeat with n = lower to upper
    add n to sum
  end repeat
  answer sum
end mouseUp
```

2. The most natural approach is to use `repeat down to`.

```
on mouseUp
  put the number of words in card field "Words" into wordnum
  repeat with theword = wordnum down to 1
    put word theword of card field "Words" into msg box
    wait 30 ticks
  end repeat
end mouseUp
```

3. You can use a `repeat until` loop.

```
on mouseUp
  repeat until the shiftKey is down
    play "boing" tempo 500 49 + random(31)
  end repeat
end mouseUp
```

   You could also use `repeat while the shiftKey is up`. Keep in mind that the `play` command runs asynchronously. That means that once the loop detects the Shift key in the down state, it doesn't send any more `play` commands, but all those `play` commands sent prior to then still get executed, so you may have to wait a bit for the sound to end after you press the Shift key down.

4. Use nested loops. The outer loop detects the Shift key and sets the tempo and the starting and ending notes. The inner loop runs the `play` command.

```
on mouseUp
  repeat until the shiftKey is down
    put random(400) + 300 into speed
    put random(59) + 11 into start
    put start + random(8) into quit
    repeat with note = start to quit
      play "boing" tempo speed note
    end repeat
  end repeat
end mouseUp
```

Chapter 11:

1. Here are the essentials:

```
on mouseUp
   put the number of words in card field 1 into wordcount
   put length(card field 1) into charcount
   put charcount / wordcount into the message box
end mouseUp
```

2. Here are the essentials:

```
on mouseUp
   put the number of words in card field 1 into wordcount
   put empty into charcount
   repeat with n = 1 to wordcount
      add length(word n of card field 1) to charcount
   end repeat
   put charcount / wordcount into the message box
end mouseUp
```

3. Here's one possible implementation, which assumes the field is called Buttons.

```
on mouseUp
  push card
  put 0 into buttoncount
  put empty into buttonlist
  repeat with bkgrnd = 1 to the number of backgrounds
    go to background bkgrnd
    repeat with bkbutn = 1 to the number of background buttons
      put the short name of bkgnd button bkbutn & ", background"¬
      && bkgrnd & ", button" && bkbutn & return after buttonlist
      add 1 to buttoncount
    end repeat
  end repeat
  pop card -- return to original card for writing into field
  put buttoncount && " BACKGROUND BUTTON(S)" & return into¬
  card field "Buttons"
  put buttonlist after card field "Buttons"
  push card
  put 0 into buttoncount
  put empty into buttonlist
  repeat with cardnum = 1 to the number of cards
    go to card cardnum
    repeat with butn = 1 to the number of card buttons
      put the short name of card button butn & ", card" &&¬
      cardnum & ", button" && butn & return after buttonlist
      add 1 to buttoncount
    end repeat
  end repeat
  pop card
  put buttoncount && " CARD BUTTON(S)" & return after¬
  card field "Buttons"
  put buttonlist after card field "Buttons"
end mouseUp
```

Chapter 12:

1. Place this script into the button:

```
on mouseEnter
  set the name of me to item 1 of the long date
```

continued

```
end mouseEnter

on mouseLeave
  set the name of me to "Day of Week"
end mouseLeave
```

2. Create a button called Short Eurodate and give it this script:

```
on mouseDown
  convert the long date to dateItems
  set the name of me to item 3 of it & "/" & item 2 of it & "/"¬
  & char 3 of item 1 of it & char 4 of item 1 of it
end mouseDown

on mouseUp
  set the name of me to "Short EuroDate"
end mouseUp
```

This script uses the last two characters of the year to show the date in the format 25/12/91. By using `item 1 of it` rather than the longer expression in the script, you would get the format `25/12/1991`.

3. Create a button called `How long?` and give it this script:

```
on mouseDown
  global pooh
  put the ticks into pooh
end mouseDown

on mouseUp
  global poo
  put the ticks - pooh && "ticks" into the message box
end mouseUp
```

Try seeing how close you can come to estimating 60 ticks.

4. Modify the script so that it reads as follows:

```
on openStack
  global startStack
  put the seconds into startStack
  put "Stack opened:" && the time & "," && the abbrev date &¬
  return after card field "Log Field" of card "Log Card"
end openStack

on closeStack
  global startStack
  put the seconds into now
  put (now - startStack) div 60 into mins
  if now mod 60 >= 30 then add 1 to mins
  put "Stack open for" && mins && "minutes" & return after card¬
  field "Log Field" of card "Log Card"
  put "Stack closed:" && the time & "," && the abbrev date &¬
  return after card field "Log Field" of card "Log Card"
end closeStack
```

The expression (now - startStack) div 60 gives the truncated time in minutes. That is, 119 seconds is truncated to 1 minute. The expression now mod 60 represents the remainder when the seconds are divided by 60. If this remainder is 30 or greater, add 1 minute to round up to the nearest whole minute.

Chapter 13:

1. Note that to get numbers in the range 0-10, we use random(11) - 1. Just using random(10) produces numbers in the range 1-10. One would expect the average to be 5, but random numbers being what they are, the actual average fluctuates from one trial to the next.

```
on mouseUp
  put random(11) - 1 into list
  repeat for 50 times
    put "," & random(11) - 1 after list
  end repeat
  put average(list) into the Message box
end mouseUp
```

2. Here's one approach:

```
on mouseUp
  put "$100,000 loan, 30-year mortgage" & return into ¬
  card field "Mortgage"
  put 100000 into loan
  put 360 into months -- monthly payments
  repeat with n = 0 to 12
    put 0.08 + n * 0.0025 into rate -- decimal fractions
    divide rate by 12 -- convert to monthly rate
    put loan/annuity(rate,months) into payments
    set numberFormat to "0.00"
    put payments * months into totalCost
    put "rate" && 1200 * rate & "%:" && "$" & payments && ¬
    "monthly for a total of $" &totalCost ¬
    & return after card field "Mortgage"
  end repeat
end mouseUp
```

Note the expression 1200 * rate. This comes from multiplying by 12 to get the yearly rate and by 100 to get a percent.

3. Try this:

```
on mouseUp
  set the loc of me to 256,170
  repeat with n = 1 to 53
    put 3 * n into radius -- increase the radius each loop
    put n * pi/5 into angle
    put 256 + radius * cos(angle) into x
    put 170 + radius * sin(angle) into y
    beep
```

continued

```
    set the loc of me to round(x),round(y)
  end repeat
end mouseUp
```

4. One way is to use the `exp2()` function:

```
on mouseUp
  ask "How many years?"
  if it is empty then exit mouseUp
  put 2 * exp2(it * 4) into rabbits
  answer rabbits && "rabbits after" && it && "years"
end mouseUp
```

Chapter 14:

1. Here's one solution:

```
function diffOverSum x,y
   return (x - y)/(x + y)
end diffOverSum
```

2. This will do it:

```
function midString str, m, n
  if m < 1 then put 1 into m
  if n < m then return empty
  if m > length(str) then return empty
  if n > length(str) then put length(str) into n
  put char m to n of str into newstring
  return newstring
end midString
```

3. Here's a solution:

```
function hypotAng x, y
  put sqrt (x*x + y*y) into side
  put atan (y/x) * 180 / pi into angle
  return side & "," & angle
end hypotAng
```

4. Here's one method:

```
function reverseWord str
  put the number of words in str into size
  if size is 0 then return empty
  put word size of str into rstring
  repeat with n = size - 1 down to 1
    put space & word n of str after rstring
  end repeat
  return rstring
end reverseWord
```

This script puts the final word of `str` into `rstring`. After that, it prefixes each word with a space before adding it. If `size` turns out to be 1, HyperTalk skips the `repeat` loop because the loop limits are in the wrong order.

5. Here's how:

```
on dialnote
  if the paramcount < 2 then
    answer "Use this way: dialnote <note> <count>"
    exit dialnote
  end if
  repeat for param(2) times
    dial param(1)
  end repeat
end dialnote
```

6. Place the following handler into the card script:

```
on mouseDown
  hide the target
  wait 10 seconds
  show the target
end mouseDown
```

Chapter 16:

1.
```
on mouseUp
  if the commandKey is down then
    doMenu "Bkgnd Info..."
  else
    if the optionKey is down then
      doMenu "Card Info..."
    else
      doMenu "Stack Info..."
    end if
  end if
end mouseUp
```

2.
```
on mouseUp
  if the commandKey is down then
    if the shiftKey is down
    then edit script of background number of this background
    else doMenu "Bkgnd Info..."
  else
    if the optionKey is down then
      if the shiftKey is down
      then edit script of card the short name of this card
      else doMenu "Card Info..."
    else
      if the shiftKey is down
      then edit script of stack the short name of this stack
      else doMenu "Stack Info..."
    end if
```

```
  end if
end mouseUp
```

3. a. scroll setter

```
on mouseDown
  global scrollNum
  repeat while the mouse is down
    if the optionKey is down
    then subtract 1 from scrollNum
    else add 1 to scrollNum
    get the id of target
    set the name of button id it to scrollNum
  end repeat
end mouseDown
```

b. big up scroll

```
on mouseDown -- "up" button
  global scrollNum
  get the scroll of card field the number of card fields
  repeat forever -- while the mouse is down
    subtract scrollNum from it
    if it < 0 then exit mouseDown
    set the scroll of card field the number of card fields to ¬
    it
  end repeat
end mouseDown
```

c. big down scroll

```
on mouseDown -- "down" button
  global scrollNum
  get the scroll of card field the number of card fields
  repeat forever
    put it
    add scrollNum to it
    if it > 312 then exit mouseDown
    set the scroll of card field the number of card fields to ¬
    it
  end repeat
end mouseDown
```

d. small up scroll

```
on mouseDown -- "up" button1
  global scrollNum
  get the scroll of card field the number of card fields
  repeat while the mouse is down
    subtract scrollNum from it
    if it < 0 then put 0 into it
    set the scroll of card field the number of card fields to ¬
    it
  end repeat
end mouseDown
```

e. small down scroll

```
on mouseDown -- "down" button1
  global scrollNum
  get the scroll of card field the number of card fields
```

```
    repeat while the mouse is down
      put it
      add scrollNum to it
      set the scroll of card field the number of card fields to ¬
      it
    end repeat
  end mouseDown
```

Chapter 17:

1.
```
on mouseUp -- handler to do custom screen blackout
  choose rectangle tool
  repeat with n = 1 to 114
    drag from 1,(n * 3)-3 to 511,(n * 3)-3
  end repeat
  repeat with n = 114 down to 1
    drag from 1,(n * 3) -1 to 511,(n * 3) -1
  end repeat
  repeat with n = 1 to 114
    drag from 1,(n * 3) -2 to 511,(n * 3) - 2
  end repeat
  choose browse tool
end mouseUp
```

2.
```
on mouseUp -- handler to do custom screen blackout 2 pass
  choose rectangle tool
  set lineSize to 3
  repeat with n = 1 to 57
    drag from 1,(n * 6)-2 to 511,(n * 6)-2
  end repeat
  repeat with n = 57 down to 1
    drag from 1,(n * 6) -5 to 511,(n * 6) -5
  end repeat
  choose browse tool
end mouseUp
```

Chapter 18:

1. The following method uses the Version 1.2 enhancements. If you don't have them, you can replace the coordinates with numerical values.

```
on openCard
  set the userlevel to 3
  set the editBkgnd to true
  set the powerKeys to true
  put 512 - width of pattern window into x
  set the loc of tool window to x,30
  set the loc of pattern window to x,bottom of tool window + 11
  set the visible of tool window to true
```

continued

```
  set the visible of pattern window to true
end openCard
```

We use the width of the pattern window rather than the tool window because the pattern window is slightly wider. The offset of 11 for locating the pattern window vertically takes into account that the `location` property doesn't include the grab bar on top of the window.

2. Here's one possibility:

```
on mouseUp
  choose select tool
  set editbkgnd to true
  doMenu "Select All"
  doMenu "Clear Picture"
  set editbkgnd to false
  doMenu "Select All"
  doMenu "Clear Picture"
  choose browse tool
end mouseUp
```

3. You can do this:

```
on mouseUp
  choose brush tool
  put 10 into x0
  put 170 into y0
  repeat with n = 1 to 11
    put x0 + 40 into x
    set the dragSpeed to 50 * random(20)
    drag from x0,y0 to x,y0
    put x + 1 into x0
  end repeat
  choose browse tool
end mouseUp
```

4. This certainly catches the eye:

```
on mouseUp
  put "1,2,3,4,none,busy,arrow,hand" into choices
  if the version < 1.2 then put 4 into limit
  else put 8 into limit
  repeat until the optionKey is down
    repeat with n = 1 to limit
      set the cursor to item n of choices
    end repeat
  end repeat
end mouseUp
```

Chapter 19:

```
on mouseUp -- Import button
```

continued

```
  put "ImportData" into fileName
  open file fileName
  repeat forever
    read from file fileName until return -- get a record
    if it is empty then -- if out of data
      doMenu "Delete Card" -- kill card made below
      exit repeat -- skip to bottom
    end if

    open file "Record" -- make a storage file
    write it to file "Record" -- put record into file
    close file "Record" -- close file and reopen for reading
    open file "Record"
    repeat with i = 1 to the number of fields
      read from file "Record" until tab -- get a field
      put it into field i -- put field data into HC field
    end repeat
    doMenu "New Card" -- make a card for the next record
    close file "Record" -- close storage file
  end repeat
  close file fileName -- close main file
end mouseUp
```

Index

A

`Abbreviated date` conversion modifier, 389
`Abs()` function, 179, 401–402
Accidentals, musical, 469–470
Accuracy of calculations, 165, 185
Active menu items by levels, 490
Addition and `add` command 134, 164–165, 168–170
`After` modifier, 46
Alerts, 13, 120
Alignment of text, 533
Allocation of memory, 616
Alternative syntax with `if` statements, 292–293
Ampersands (&) for joining strings, 48, 305–306, 346–348
Amplitude of sine waves, 464–466
`And` operator, 282–284, 285–286, 288–290
Animation, 552–553
`Annuity()` function, 409–412
`Answer` command, 120–123, 125, 257–259
`Any` descriptor, 95
APDA (Apple Programmer's and Developer's Association), 13, 23–24, 624
Applications and HyperCard, 4–5
Applicator tools, simulation of, 525–534
Arctangent, function for, 418–419
Arguments, function, 179–183
    with user-defined functions, 433
    with XCMDs and XFCNs, 615
    See also Parameters
Arrow keys, property for, 562–563
`ArrowKey` message, 210, 259, 262–262
ASCII (American Standard Code for Information Interchange)
code and functions, 352–355
    character set of, 346
`Ask` command, 123–125, 251–254
`Ask password` command, 254–257
Assignment of variables, `put` for, 130, 275
Asterisks (*) for multiplication, 165
`Atan()` function, 418–419
AtDrvrVersNum with XCMDs and XFCNs, 626
Atkinson, Bill
    and cards, 11, 60
    and compilation of HyperCard, 13
    and development of HyperCard, 5–8
    on flexibility of HyperCard, 222–223
    and keyboard alternatives, 21
Audio. *See* Sound
Authoring level, 32–33, 561
Authoring system, HyperCard as, 6–7
`AutoHilite` property, 505–506
Automatic formatting of scripts, 40, 294
`Average()` function, 180, 402, 404

B

B for musical flats, 468–470
`Background` command, 493, 549
Background layer, 523
Backgrounds
    copying of, 122

Backgrounds—cont
objects in, 65–67, 73
properties of, 503–504
relationship of, to other objects, 78–79
simulation of menu items for, 493
switching to, 21
toggle for, 569
Backing up of stacks, 491
Bar character (¦) in command syntax, 46
`Barn door` visual effect, 115–116
Barret, Bruce, and Groupies script, 52
Base 2, exponent function for, 420–421
Base 10, logarithms for, 423
Bases for exponents, 421
BASIC compared to HyperTalk, 16–17
`next repeat` command, 329
`play` command, 474
subroutines, 92
`trunc()` function, 183
variables, 148
`Beep` command, 37, 309
`Before` modifier, 46
Bifurcations, 193
Bit-mapped images, 64, 523
compression of, 8
`Bkgnd Info` command, 493
`Black` modifier, 107, 109
Blind Typing box, 33, 45
`BlindTyping` property, 563–565
Blocks
with `if` statements, 293
memory, 616
Body of loops, 310
Boing voice, 42–43, 103, 466
sound effects using, 481
Bold text, 533
Books for HyperCard, 25–27
Boolean expressions, 241, 272–274
compared to messages, 327
and logical expressions, 282–290
and relational expressions, 274–280
use of, 280–282
`BoolToStr` callback, 620
Borders, 534–535
`Bottom` property, 510, 584–585
`BottomRight` property, 510, 584–585
Brackets ([ ]) in command syntax, 50
Branching statements. *See* `If then` control statements
Brand, Stewart, 24
`Bring Closer` command, 81, 496–497
Browse tool, 23, 500
for visual effects, 107
Browsing level, 7, 32, 561
`Brush Shape` command, 550
Brush tool, 527–530
Bucket tool, 531–532
Bugs, Hypercard, 647–648
See also Debugging
Built-in looping, 309
Built-in XCMDs, 616–617
Bundling of HyperCard, 7–8
Busy cursor, 567
Button, mouse, 37
functions for, 239–245
messages for, 202, 205–206
Button tool, 23, 50, 500
Buttons, 61–63
in answer boxes, 120
as controls, 12
and fields, 77
flipping and rotating of, 543–544
messages with, 70–71, 213
properties of, 504–511
relationship of, to other objects, 79
removing icon from, 567
scripts on, 42–43, 575
simulation of menu items for, 495–498
word counting with, 362–363

C

C language
compared to HyperTalk, 16–17
with XCMDs and XFCNs, 627–628
Calculator
Message box as, 163–164
script for, 175–176, 180–181
Callbacks with XCMDs and XFCNs, 618–622
Calling of functions, 432
`CantDelete` property, 645
`CantModify` property, 576, 645
Capitalization, ChangeCase XFCN for, 630–643
`Card` modifier, 107, 109

Card Info command, 493–494
Cards, 11, 59–62
  copying of, 52, 122
  creation of, 21
  elements of, 67
  messages with, 71, 213–214
  objects on, 65
  properties of, 503
  relationship of, to other objects, 78–79
  scripts on, 42
  simulation of menu items for, 493–494
Carets (^) as exponent operator, 174
Carriage return as delimiter, 596
Case sensitivity, 38
  with encryption, 256
  with test expressions, 365
CD ROMS, properties for, 645
Centered property, 535–536
ChangeCase XFCN, 630–632
  source code for, 632–643
Chaos theory, 191–194
Characters
  ASCII codes for, 352–355
  chunks of, 355–356
  range notation for, 438
  in strings, 346
CharToNum() function, 353
Check boxes, 12
Checkerboard visual effect, 110–111, 282
Choose command, 118–119, 647–648
  simulation of, 500, 524–549
Chunks, 355–359
  counting of, 360–362
  note, 476
  selection of, 370–371
  storage of, with files, 603–604
Circles and circular motion, 416–418, 536–540
Clear command, 495
Clear key, 164
Clear Picture command, 549
Clicking of mouse and click command, 37, 241–243
  with brush tool, 527, 529–530
  with Command key, 207
  simulations of, 243–245, 511–512
  with text tool, 532
Click at command, 243–245, 499
Clipboard, 21, 23, 493, 495, 593
Close boxes, 11
Close file command, 598
CloseBackground message, 215, 499
CloseCard message, 214–216, 499
CloseField message, 212–213
CloseStack message, 145, 215–216, 218, 499
Closing of files, 598
Combining of messages, 214–218
Command keys
  clicking with, 207, 243
  Command-+, 497
  Command-., 256, 482
  Command-1, 21
  Command-2, 21
  Command-3, 21
  Command-4, 21, 33
  Command-=, 21, 65
  Command-?, 218
  Command-A, 39
  Command-B, 21, 65
  Command-C, 21, 23, 39
  Command-F, 39
  Command-G, 39
  Command-H, 33, 39
  Command-(hyphen), 21, 497
  Command-M, 21, 22, 44
  Command-O, 21
  Command-P, 39, 493
  Command-Q, 21
  Command-S, 21
  Command-space, 22, 61
  Command-Tab, 22–23, 50
  Command-V, 21, 23, 39
  Command-X, 21, 23, 39, 249–251, 500
Command messages, 220–223
CommandKey constant
  and brush tool, 527
  with drag command, 514, 646
  with lasso tool, 547
  with select tool, 541–542
  with type command, 265
Commands
  and Message box, 44–45
  modification of, 452–453
  See also User-defined commands
Commas (,) with multiple parameters, 179, 434, 445
Comments, 43–44

Compact Stack command, 491
Comparisons of strings, 282
callback for, 622
Compilation, 13
Composite chunks, 357
Compound() function, 407–409
Compression, bit-map, 8
Compuserve, 24
Concatenation of strings, 48, 305, 346–348
Conditional loops, 320–327
Constants
in Boolean expressions, 274
as strings, 190, 343–345
text, 348–351
as values, 134–136, 167–168
Containers, 46, 129–131
fields as, 138–141
Message box as, 155–157
storage of, 616–617
the selection as, 152–155
variables as, 141–152
Contains relational operator, 277, 364–366
for searching, 373–375
Control key and controlKey message, 210, 259, 263–264
for termination of loops, 333
Control Panel command, 499
Control statements. *See* If then control statements
Controls, 12
Conversion
of dates, 388–392
of degrees and radians, 413, 453–454
of elapsed time, 395–396
of strings, 134–135, 282, 619–620
of time formats, 384–385
Convert command, 388–392
Coordinates
mouse, 202–205, 231–239
of screen, 582
Copy command, 21, 23, 495
Copy Card command, 493–494
Copy current background option, 34
Copying, 21, 23, 39, 542
of cards, 52, 122, 493–494
by dragging, 513
Cos() function, 412–413, 415–418
Counting functions, 359–363
Creation information for stacks, 392
Crosshatch character (#)
in formatting, 189–190
for musical sharps, 468–470
Cursor, 10, 236
and arrow keys, 33, 562–563
and messages, 210
Cursor property, 565–567
Curve tool, 539
Custom messages, 225–226
Customization of keyboards, 211–212
Cut Card command, 493
Cutting and cut command, 7, 21, 23, 39, 495, 542
Cutting and pasting level, 7

D

Darken command, 22, 545
Data checking, 301
repeat until loops for, 323–324
DateItems format, 385, 388–390
Dates and date() function, 381–385
conversions of, 388–392
European, 384–385
time information with, 387–388
Day of week, 383
conversion functions for, 389
script to find, 396–397
Debugging
and arithmetic commands, 172
comments for, 43
and global variables, 157
script tool for, 177
of XCMDs and XFCNs, 624
Declaration of variables, 143
Decrement script, 176
Defaults with answer boxes, 120
Defining of user-defined functions, 431
Degrees, conversion of radians to, 413, 453–456
Delete Card command, 493–494
Delete Stack command, 491
DeleteBackground message, 214–215
DeleteButton message, 213
DeleteCard message, 214
DeleteField message, 212
DeleteStack message, 215

Delimiters, 596
Desk Accessories, doMenu access to, 117, 490–491, 498
Desktop metaphor, 11
Destinations
with arithmetic commands, 169–171
chunks as, 358–359
for commands, 46
containers as, 138
nonexistent, 455
Developer's Technical Support, 623
Dial message, 221–222
Dialog boxes, 12–13, 549–550
Dials, 12
DiskSpace() function, 648
Dissolve effect, 107–108
Div operator, 173
for elapsed time conversions, 395–396
Division and divide command, 165, 168–169, 171–173
Document icon, 67
Documentation, 167
DoList XFNC, 125
DoMenu command and messages, 116–118, 220–221, 489–491, 495
alternative to, 500
cutting, copying, and pasting with, 542
"Done" string, 484–485
Dotted musical notes, 467, 473
Double-clicking
for selection, 369
with Shift key, 22
simulation of, 243
Double indirection, 616
Down constant, 135, 239, 249
with arrowKey message, 262
with visual effects, 112
Down to with repeat structures, 314–316
Drag command, simulation of, 512–514
DragSpeed property and dragging, 567–569
with applicator tools, 525–530
with manipulator tools, 541–549
with shape tools, 534–540
shift and option keys with, 22, 514, 539
DTS (Developer's Technical Support), 623
Duration parameter, 471–473

E

Edit Pattern command, 550
EditBkgnd property, 569–570
Editing
of scripts, 35, 38–41
of stacks, 574–575
of text, 13–14
Eight constant, 134, 167
Elapsed time, 392–396
Elements, card, 67
Else with if statements, 294–298
Empty constant, 124, 135–136, 349
with select, 372–373
Enabling technology, HyperCard as, 3–4
Encryption for passwords, 254–257
End commands, 37
end if, 292–294
end repeat, 310–311
Enter key and enterKey message, 38–39, 209, 259–261
EnterInField message, 210
Entry condition loops, 325
Entry point of objects, 70, 454
EnvironsVersion with XCMDs and XFCNs, 625
Equal sign (=) with relational expressions, 274–278
Equality, testing for, 274–276
Equally tempered musical scale, 467
Eraser tool, 548–549
Erasing with into, 48
Errors
checking for, 173, 456
messages for, 172–173
See also Debugging
Espinosa, Chris, 8
European data and time, 384–385
EvalExpr callback, 619
Event-driven programs, 14
Exclusive or, 285
Exit commands, 301–302
exit repeat, 321, 328–329, 333
exit to HyperCard, 302
Exiting of scripts, 331–332
ExitToShell with XCMDs and XFCNs, 625
Exp( ) function, 419–422
Exp1( ) function, 421
Exp2( ) function, 420–421

Exponent operator, 174
Exponential functions and behavior, 419–424
Exponential notation, 185–186
`Export Paint` command, 549
Expressions
  compared to messages, 327
  functions in, 434
  with math commands, 169–170
  as parameters, 440
  See also Boolean expressions
Extended text, 533
`ExtToStr` callback, 620

F

F for filling of selections, 22
`False` constant, 135, 168, 241, 272
  See also Boolean expressions
`Fast` modifier, 107
Field tool, 50, 500
Fields, 61, 63–64
  and buttons, 77
  as containers, 138–141
  for files, 603–604
  locked, and `the selection`, 478
  messages to, 71, 207–214
  properties for, 504–511
  relationship of, to other objects, 79
  selection of, 370
  simulation of menu items for, 495–497
  for XCmdBlock, 628–629
FIFO (first in first out) storage, 596–597
Files, 14, 593
  closing of, 598
  examples using, 599–609
  locked, 575–576
  opening of, 594–595
  reading of, 596–598
  storage of chucks on, 603–604
  writing to, 595–596, 599–600
  with XCMDs and XFCNs, 627
`Fill` command, 22, 545
Filtered formatting of numbers, 187–190
Financial functions, 407–412
`Find` command, 39–40, 373, 455, 498, 645, 647
`First` descriptor, 95
First in first out storage, 596–597
`Five` constant, 134, 167
Fixed iteration loops, 309–316
`Flash` XCMD, 616
Flats, musical, 104, 468–470
`Flip Horizontal` command, 22
`Flip Vertical` command, 545
Flipping of buttons, 543–544
Flow charts, 291
Folders, 14, 34, 93
Fonts, 38
  and character alignment, 355
  properties of, 507
  for text, 533–534, 549
`For` with `repeat` loops, 310
Foreground
  objects in, 65
  switching to, 21
`Forever` with `repeat` loops, 331–334
Formal parameters, 433
Formatting
  of files, 594
  of numbers, 187–190
  of scripts, 40
`Formfeed` constant, 349, 351
`Four` constant, 134, 167
Fractional exponents, 420
Fractional part of number, function to find, 403
`Freesize` property, 504
Frequency of sound, 465–466
`From` keyword, 171
`Function` keyword, 431
`FunctionKey` message, 210, 259, 263
Functions, 136–137, 177–179, 645
  compared to properties, 235–236
  modification of, 453–454
  overriding of, 429
  See also Math functions; User-defined functions
Fundamental frequencies, 465

G

`Get` command, 502
  with `it`, 149–152
`GetDocs` XCMD, 617

GetFieldByID callback, 621
GetFieldByName callback, 621
GetFieldByNum callback, 621
GetGlobal callback, 620
GetNextEvent with XCMDs and XFCNs, 627
Global command and global variables, 143–146, 157
with XCMDs and XFCNs, 624
Global properties, 559
User Preference, 560–577
window, 577–586
Go commands, 41, 91–93, 498
go back,, 95
and visual effects, 107
Go menu, 33
Graphics, 9–10, 61, 64–65, 523–524
applicator tools, 525–534
and dialog boxes, 549–550
and hiding of pictures, 551–554
manipulator tools, 541–549
script using, 554–556
shape tools, 534–541
Gray modifier, 107, 109
Greater than sign (>)
in command syntax, 46
as relational operator, 277–278
Grid property, 540–541
Groupies script, 52, 122
Growth, exponential, 423–424

H

H for Flip Horizontal command, 22
Handlers, 36–37
generalization of, 86
libraries of, 225–226
for messages, 18
sharing of, 73, 204–205
user-defined commands as, 441
Handles with XCMDs and XFCNs, 615–617
Hands On Macintosh, 25
Harmonic frequencies, 465
Harpsichord voice, 43, 103, 466
HasColorQD field with XCMDs and XFCNs, 626
HasFPU field with XCMDs and XFCNs, 626
Heap, 615–616
with XCMDs and XFCNs, 626–627
Height property, 510, 584
Help message, 218
Hertz, Heinrich, 465
Heterogeneous stacks, 66–67
HFS (Hierarchical File System), 89, 594
Hiding and hide command, 83, 550, 645
for fields, 141
of objects, 82–85
of pictures, 551–554
for windows, 583
Hierarchical File System, 89, 594
Hierarchy
of objects, 70–78, 199–200
of user-defined commands, 450–451
High-tempo sound effects, 482
Highlighted characters as the selection, 152
Hilite property, 506
Holders, 604
Home stack, 33
messages with, 73–74
Homogeneous stacks, 65–66
Horizontal position of mouse, 232–233, 238–239
Hot spot, mouse, 236
Human Interface Guidelines: The Apple Desktop
Interface, 15
HyperCard, 3–5
history of, 5–8
relationship of, to HyperTalk, 41–42
updates of, 458
"HyperCard Developer's Toolkit," 623
HyperLink, 25
HyperMedia, 5–6, 84
HyperXCmd.p callback, 618
Hyphens (-) for comments, 43–44
Hypotenuse, script to find, 434–435

I

I for Invert command, 22
Icon property, 505
Icons, 12
document, 67
removal of, from buttons, 567

ID (identification) numbers, 41, 82, 88
Id property, 507
Idle messages, 146, 217
and cursor property, 565
and lockRecent property, 573
and lockScreen property, 571
and time updates, 386
If then control statements, 271–272, 290–294
if then else, 294–298
if then else if then else, 298–300
with modifier keys, 251
nesting of, 299–302
Image modifiers, 107, 109
Import Paint command, 549
In keyword, 90
Inclusive or, 285
Increment script, 176
Indentation of scripts, 40, 294
INF error message, 172
Inheritance of properties, 18
Initialization of variables, 404–405
Insertion point, 264
Inside Macintosh, 622–623
Instability zones in chaos theory, 191, 43
Installation of ChangeCase XFCN, 637
Interception of commands, 452–453
Interest, functions for, 407–412
Interfaces, Macintosh, 9
International stacks, support for, 570, 646
Into modifier, 46
erasing and replacing with, 48
Inverse modifier, 107, 109
Invert command, 22, 547
Investment functions, 407–412
Iris visual effect, 115–116
Irrational numbers, 168
Is relational operators, 17, 277
is in,, 277, 364–366, 373–375
is within, 277, 279–280, 510–511
It variable, 124, 136
with answer command, 259
with get command, 149–152
Item chunks, 355

J

Joining of strings, 48, 305, 346–348

K

Kaehler, Carol, 8
Kaehler, Ted, 8
script by, 575
Kamins, Scott, 8
Keep command, 549
Keyboard
as alternatives to menus, 21–22, 645
functions for, 249–251
global properties for, 562–565
messages from, 209–212, 259–264
and mouse buttons, 240
and Power Keys, 33
simulation of, 264–265
user interface functions for, 251–259
KeyBoardType with XCMDs and XFCNs, 626
Keys for encryption, 255
Kreitman, Kristee, 8

L

Lacy, Sioux, Groupies script by, 52
Language property, 570
Language vocabulary, 15
Largest numbers, function to find, 403–404
Lasso tool, 545–548
with select tool, 541–542
Last descriptor, 33, 95
Last in first out storage, 596–597
Layers, object, 65, 523–524
Left constant
with arrowKey message, 262
with special effects, 112
Left property, 510, 584–585
Left Arrow key, 33
Length of strings and length() function, 359–360, 362
callback for, 619
Less than sign (<)
in command syntax, 46
as relational operator, 277–278
Levels, user, 7, 20–21, 31–33, 490, 559
Libraries, handler, 225–226
LIFO (last in first out) storage, 596–597
Lighten command, 545

Line chunks, 355
`Line Size` command, 549
Line tool, 526–527
`Linefeed` constant, 349, 351
Lines, selection of, 370–371
`LineSize` property, 526–527
  and rectangle tool, 534
  with regular polygon tool, 539
Lissajous figures, 587–588
`Ln()` function, 421–423
Local variables, 143, 146–149, 152
Location of stacks, 34–35
`Location` property, 231, 504–505
  with `click` command, 511
  for windows, 578–581
`Lock screen` command, 572, 645
Locked fields, 207–209, 212
  and `the selection`, 478
Locked files, 575–576
`LockMessages` property, 573–575
`LockRecent` property, 572–573
`LockScreen` property, 570–572
`LockText` property, 63, 207–209, 212, 507
Logarithmic functions, 421–423
Logical constants, 168
Logical expressions, 282–290
Logical operators with test expressions, 322–323
Logo, 7
`Long date` conversion modifier, 389
Long lines
  with `if` statements, 292–293
  option-return for, 40–41
`Long time` conversion modifier, 389
`LongToStr` callback, 620
Loops
  conditional, 320–327
  fixed iteration, 309–316
  modification of, 327–334
  nesting of, 316–320
Lowercase letters, 38
  and ASCII codes, 352–353
  with comparisons, 282
  with testing, 365

M

MachineType constants with XCMDs and XFCNs, 625
Macintosh, principles of, 9–15
Macintosh II, visual effects on, 108, 524
"Macintosh Technical Notes," 623
MacPaint environment, 523
"MacTutor," 623
Magazines for HyperCard, 25, 623
Maller, Steve
  ResCopy by, 13
  scrolling file boxes by, 14
Manipulator tools, simulation of, 541–549
Math
  coprocessor for, with XCMDs and XFCNs, 626
  error messages for, 172–173
  operators for, 163–166, 173–174, 183–184
Math functions
  exponential, 419–422
  financial, 407–412
  general, 401–407
  logarithmic, 421–423
  trigonometric, 412–419
`Max()` function, 403–404
`Me` function, 85–86, 95
Memory
  for graphics, 10
  heap, 615–616, 626–627
  with XCMDs and XFCNs, 615
Memory Manager, 616
Menu bars, 11, 22, 61
Menus, 11
  for dialog boxes, 549–550
  keyboard with, 21–22, 645
  messages from, 220–221
  simulation of, 489–499
Message box
  and blind typing, 33, 45, 563–565
  calculations in, 163–164
  commands in, 37
  as container, 155–157
  corrections in, 133
  hiding of, 21–22, 44
  messages in, 226
  `put` command with, 47–48
  as scripting control panel, 44–45

Message box—cont
sending messages from, 319
the with, 578
for viewing hidden objects, 85
Messages, 17–19, 37, 43, 199–201
combining of, 214–218
command, 220–223
compared to expressions, 327
custom, 225–226
to fields, 207–214
from keyboard, 209–212, 259–264
locking of, 573–575
in Message box, 226, 319
mouse, 202–206
and object hierarchy, 70–74
from objects, 212–214
parameters for, 201–202
passing of, 74–78, 218–220
reporting, 202–206
sending of, 223–225, 319
Middle descriptor, 95
Min() function, 403–404
Minus sign (–) for subtraction, 165
Mod operator, 174
for elapsed time conversions, 395–396
Modal dialog boxes, 12–13, 38, 120, 123
Modification
of commands, 452–453
of functions, 453–454
of stacks, date of, 392
Modifiers, 46
Month, functions for, 383, 389
Mortgages, function for, 411–412
Mouse, 10
button functions for, 239–245
clicking of, 37
hot spot of, 236
messages from, 202–207
position of, 231–239, 280–281
MouseDown message, 37, 205–208, 327
MouseEnter message, 203–204, 208
MouseLeave message, 203–204, 208
MouseStillDown message, 206, 208
MouseUp message, 37, 201, 206–208
MouseWithin message, 203–204, 208
Movie special effects, 111–114
Msg box. *See* Message box
MultiFinder, memory requirements with, 10
Multiple parameters
functions with, 434–436
with user-defined commands, 445–446
Multiple property, 537–538
Multiplication and multiply command, 165, 168–169, 171
MultiSpace property, 539
Music. *See* Sound

N

Name property, 505, 507
Names
for cursors, 566
numbers for, 142–143
for objects, 62, 79–89
option characters in, 39
for stacks, 34–35
for variables, 130–131, 142
NAN error message, 172–173, 406
Navigation, 21, 90–95, 498, 647
Negative exponents, 420
Negative numbers
square root of, 405–406
truncation of, 402
Nesting
of if statements, 299–302
of loops, 316–320
Networks for HyperCard, 24
New Background command, 493
New Button command, 496
New Card command, 21, 493–494
New Field command, 496
New Stack command, 34, 491
NewButton message, 213
NewField message, 212
Next descriptor, 95
Next repeat command, 321, 329–331
Nine constant, 134, 167
Nonprinting characters, 346, 354
Nonrelocatable memory blocks, 616
Not logical operator, 279, 286–290
Not equal to operator, 277
Note fields, pop-up, 84
NoteName parameter, 469
Notes, musical, 104, 465–470
duration of, 471–473

Notes—cont
numbers for, 470
Number property, 507
NumberFormat property, 187–190, 359
Numbers
for cursors, 566
formatting of, 187–190, 359
identification, 41, 82, 88
irrational, 168
for musical notes, 470
for names, 141
operators for, 163–166, 173–174
and parameters, 436–438
range of, 185–187
relative, 81–82
and strings, 346, 359
as values, 132–134
See also Math functions
Numeric constants, 134, 167
as parameters, 441
NumtoChar() function, 353
NumToHex callback, 620
NumToStr callback, 620

O

Object-oriented programming, 15, 17–20
Objects, 17
button, 62–63
callbacks to access, 620–621
card, 59–62
as containers, 138–141
field, 63–64
graphics, 64
hiding of, 82–85
hierarchy of, 70–78, 199–200
and layers, 65–67
messages from, 74–78, 212–214
naming of, 79–89
and navigation, 90–95
ordinal descriptors for, 95
and pathnames, 89–90
properties of, 35
relationship between, 78–79
and stacks, 67–70
for user-defined functions, 431–432
See also specific objects
Octaves, 104, 467, 469, 471
Of, 90
with properties, 50
with user-defined functions, 432
with window properties, 578, 580–581
Offset() function, 366–369
for searching, 373–375
On with handlers, 37
One constant, 134, 167
Opaque layers, 65
Open file command, 594–595
Open Stack command, 21, 491
OpenBackground message, 215
OpenCard message, 214–216
OpenField message, 212–213, 216
Opening
of button scripts, 22, 575, 645
of files, 594–595
OpenStack message, 215–216, 404–405
Operating systems, 9
Operators, 17
math, 163–166, 173–174
precedence of, 183–184, 288–290
Option characters in names, 39
Option keys
with arrow keys, 562
with click at, 243
dragging with, 22, 539
as modifier, 249–251
Option-<, 278
Option->, 278
Option-return, 40–41, 292–293
Option-tab, 157
OptionKey message
and line tool, 526
with rectangle tool, 534
with select command, 546–547
with select tool, 541–542
with type command, 265
Options menu, power keys for, 564
Or operator, 284–286, 288–290
Ordinal descriptors, 95
Oscillations and trigonometric functions, 414
Outlined text, 533
Oval tool, 536–539
Overflow, 187
Override, user-defined function, 429

P

Paal, Adam, 8
Painting level, 7, 32, 561
Painting properties, 524–525
Parameters, 117, 201–202
  for functions, 453
  for messages, 262
  for user-defined commands, 444–450
  with user-defined functions, 432–438, 440–441
  See also Arguments
Parentheses ( )
  with logical expressions, 287, 290
  for operator precedence, 183–184
  with user-defined functions, 432, 434, 436
Pascal and HyperTalk, 18
`Pass` command, 74–78, 218–220
  with user-defined commands, 451–452
Passwords, 254–257
Pasting and `Paste` commands
  by dragging, 513, 542
  into hidden pictures, 554
  with keyboard, 21, 23, 39
  `Paste Card`, 493–494
  `Paste Picture`, 554
  simulation of, 495
  of text in fields, 646
`PasToZero` callback, 619
Pathnames, 594
  for stacks, 89–90
Patterns
  with bucket tool, 531
  editing of, 550
  of lines, 528
Patterns menu, 11
  power keys for, 564
Pauses, 310–311
PC version of HyperCard, 8
Pencil tool, 525–526
Periodic functions, 414
Periodic sound, 463–466
Periods (.) with musical notes, 467, 473
Pi constant, 167–168
Pictures
  hiding and showing of, 551–554
  in stacks, 384
Pilot, 6–7
Pixels, 10
  and graphics, 64
  and mouse location, 231
Plato, 6
`Play` commands, 42–44, 103–106, 466–475
  `play stop`, 482–483
Plus sign (+) for addition, 134, 164–165
Point size, 38
Pointers to memory, 616
`Polygon Sides` command and `polySides` property, 539–540, 550
Polygons, tool for, 539–540
`Pop` command, 94–95
Pop-up note fields, 84
Population, chaos theory for, 191–194
Position of mouse, 231–239, 280–281
Pound sign (#). *See* Crosshatch character
Power keys and `powerKeys` property, 22, 33, 563–565
Precedence of operators, 183–184, 288–290
Precision of calculations, 165, 185
`Prev` descriptor, 95
Printing
  power keys for menu for, 564
  `Print Card` command, 493
  `Print Report` command, 491
  `Print Stack` command, 491
  of ranges of scripts. 352
  and `return` constant, 350–351
Private Access bug, 265
Procedural languages, 18
Processor with XCMDs and XFCNs, 625–626
Programming language, HyperCard as, 6–7
Properties, 17–18, 35, 137–138
  compared to functions, 235–236
  painting, 524–525
  and pencil tool, 525
  `set` command for, 50
  simulations with, 502–511
  window, 577–586
  See also Global properties
`Protect Stack` command, 491, 576
Pseudoscientific calculator script, 175–176, 180–181
`Push` command, 94–95

Put command, 46–50, 502
for assignments, 130, 275
with fields, 138
with Message box, 155
with variables, 143

Q

Question mark (?) for overflow, 187
QuickDraw, 5
Quit HyperCard command, 498–499
Quitting and quit message, 21, 145, 218, 499
backing up during, 491
Quotation marks (")
for parameters, 447–449
for strings, 132, 343–345
and words, 355
Quote constant, 135, 349

R

Radians, conversion of degrees to, 413, 453–454
Radio buttons, 12
Radioactive decay, exponent function for, 421
Random() function, 52, 177–179, 405
Random pixel-mixing effect, 107–108
Ranges
character, 438
with chunks, 357–358
of numbers, 185–187
printing of, 352
Read command for files, 593, 596–598
Recent command and lockRecent property, 572
Rectangle property, 504–505, 509–511
for windows, 581–586
Rectangle tool, 534–535
Regular polygon tool, 539–540
Relational expressions and operators, 274–279
and test conditions, 322–323
Relative numbers, 81–82
Relocatable memory blocks, 616
Remainder operator, 174, 395–396
Repeat control structures
nesting of, 316–320
repeat for, 310–312
repeat forever, 331–334
repeat until, 320–324
repeat while, 324–327
repeat with, 312–316
Replacing data with into, 48
Reporting messages, 201–206
ResCopy XCMD, 13, 637
ResEdit, 13, 125
Reserved words
as object names, 85
as variable names, 143
Reset paint command, 526
Resource fork, 13
for XFCNs, 637
Resources
for HyperCard, 23–27
for XCMDs and XFNCs, 622–624
Rests, musical, 474
Resume message, 216–217
Return constant, 48, 135, 349–350
as white space, 355
Return key and returnKey message, 38, 209, 259–261
Return with user-defined functions, 434, 438–440
ReturnInField message, 210
ReturnToPas callback, 622
Revert command, 549
Right constant
with arrowKey message, 262
with visual effects, 112
Right property, 510, 584–585
Roots, fractional exponents as, 420
Rotating of buttons, 543–544
Round() function, 173, 181–182, 402–403
Round rect tool, 535–536
Rounded rectangle buttons, 62

S

Salt and peppering effect, 107–108
SANE (Standard Apple Numerics Environment), 185, 406
Save a Copy command, 491

Saving of global variables, 145
Scales, musical, 467, 469
ScanToReturn callback, 622
ScanToZero callback, 622
Scientific calculator script, 175–176, 180–181
Scientific notation, 185–186
Scope of user-defined commands, 441–443
Scrapbook, 498
Screen
  coordinates of, 582
  flashing of, 616
  locking of, 570–572
ScreenRect() function, 582–583, 645
Script editor, 35, 38–41
Script property, 35, 503, 505
Scripts, 17
  arithmetic operators in, 166
  in buttons, opening of, 22, 62, 575, 645
  for control structures, 303–307
  for day of week, 396–397
  editing of, 33, 38–41
  exiting of, 331–332
  for file I/O, 604–609
  for global properties, 586–588
  for graphics, 554–556
  location of, 42–43
  opening of, 507
  peeking at, 23
  printing ranges of, 352
  for simulation, 514–517
  for sound lab, 475–480
  stack, 69
  for strings, 373–375
  for user interface, 336–338
  user level for, 33, 561
Scroll property, 507–508
Scroll visual effect, 112–114
Scroll-right visual effect, 282
Scrolling bars, 11–12
Sculley, John, 7–8
Searching, 455
  in scripts, 39–40, 177, 373–376
  for strings, 619
SearchScript function, 177
Seconds, 92
  function for, 392–396
  and wait command, 121–122
Seconds conversion modifier, 388–389
Select command, 372–373, 546–548, 645
Select menu command, 21
Select tool, 541–543
Self-documentation, 167
Semilogarithmic plots, 423–424
Send command, 73, 223–226
Send Farther command, 81, 496–497
SendCardMessage callback, 618–619
Sequential files, 593
Set command, 50–52, 83
  simulations with, 502–511
SetFieldByID callback, 621
SetFieldByName callback, 621
SetFieldByNum callback, 621
SetGlobal callback, 620
Seven constant, 134, 167
Shape tools, simulation of, 534–541
Sharing of handlers, 73, 204–205
Sharps, musical, 104, 468–470
Shift key and shiftKey message
  with click at, 243
  dragging with, 22, 514
  with eraser tool, 549
  with lasso tool, 547
  and line tool, 526
  as modifier, 249–251
  with oval tool, 536
  with pasting of fields, 646
  and pencil tool, 525
  with rectangle tool, 534
  with regular polygon tool, 539
  with type command, 265
Short date conversion modifier, 389
Short time conversion modifier, 389
Show command, 83, 550, 645
  for windows, 583
Show all cards command, 309
Showing of pictures, 551–554
ShowLines property, 507
ShowName property, 505
ShowPict property, 551–553
Silence voice, 104, 466, 474
Simulation
  of applicator tools, 525–534
  of background menu items, 493
  of card menu items, 493–494
  of choose command, 500, 524–549
  of click command, 511–512

Simulation—cont
of drag command, 512–514
of field and button menu items, 495–498
of keyboard, 264–265
of manipulator tools, 541–549
with properties, 502–511
of shape tools, 534–541
of stack menu items, 491–492
of the tool function, 500–502
of type command, 499–500
Sin() function, 412–418
Sine waves, 464–466
Six constant, 134, 167
Size of text and size property, 504, 533–534, 549
Skipping within loops, 329–331
Slashes (/) for division, 165
Slides, wipe effect to mimic, 114
Slow modifier, 107
Smallest numbers, function to find, 403–404
Snd resources, 480
Sound
beep command for, 37, 309
lab for, 475–480
play command for, 42–44, 103–106, 466–475
and sound effects, 480–485
Source containers, 138
with arithmetic commands, 169–171
Space constant, 48, 135, 349
Spaces
in date formats, 384
with parameters, 180
in strings, 132, 276, 346–347
in variable names, 142
as white space, 355
Speakers. *See* Sound
Speed modifiers for visual effects, 107–110
Spray tool, 530–531
Sqrt() function, 405–407
Stability zones in chaos theory, 191
Stack Info command, 491
Stack memory, 615–616, 647
Stacks, 19, 67–70
creation of, 21, 33–35
editing of, 574–575
Stacks—cont
heterogeneous and homogeneous, 65–67
modification date of, 392
music with, 483–485
names and location of, 34–35
objects for, 73
pathnames for, 89–90
properties for, 503–504
protection of, 576–577
relationship of, to other objects, 79
simulation of menu items for, 491–492
version number in, 456–457
Standalone applications, 67–68
Standard Apple Numerics Environment, 185, 406
StartUp message, 216
Static variables with XCMDs and XFCNs, 624
StringEqual callback, 622
StringLength callback, 619
StringMatch callback, 619
Strings
callbacks for, 619–620
comparison of, 274, 278, 282
constants as, 190, 343–345
joining of, 48, 305–306, 346–348
length of, 359–360, 362
for musical notes, 467
and numbers, 134–135, 346, 359
and parameters, 436–438, 440
in relational expressions, 276–277
strings within, 363–369
and text constants, 348–351
and the selection, 369–373
as values, 132
StrToBool callback, 620
StrToExt callback, 620
StrToLong callback, 620
StrToNum callback, 620
Style of text and style property, 507, 533–534, 549
Substrings, 366–368
Subtraction and subtract command, 165, 168–169, 171
Suspend message, 145, 216–217
Sweep sound effects, 480–482
Synchronization of sound, 483–485
Syntax, 15

Syntax—cont
of commands, 46–47, 50
SysEnvirons field with XCMDs and XFCNs, 625
SystemVersion field with XCMDs and XFCNs, 625
SysVRefNum field with XCMDs and XFCNs, 626

T

Tab key and tabKey message, 209, 259–261
for script formatting, 39–40
Tabs and tab constant, 135, 349–351
as delimiter, 596
as white space, 355
Tan() function, 412, 415–416
Taxes, script for, 303–306
Tear off menus, 11
Tempo parameter, 105–106, 466–467, 473–474
Ten constant, 134, 167
Termination of loops, 328–329, 333
Test conditions, 272, 322–323
Testing
for content, 364–369
of XCMDs and XFCNs, 621
Text
in background fields, 65
chunks of, 355–359
constants for, 348–351
editing of, 13–14
and keyboard messages, 260
measuring functions for, 360–363
selection of, 23, 369–373
wrapping of, 316
Text Arrows box, 33
Text Style command, 495, 549
Text tool, 532–534
TextAlign property, 507, 533
TextArrows property, 562–563
TextFont property, 507, 533
TextHeight property, 507, 533
TextSize property, 533
TextStyle property, 507, 533
The
with Message box, 578, 581
The—cont
with user-defined functions, 432, 436, 440
with window messages, 578, 581
The abbreviated date function, 382
The clickH function, 238–239
The clickLoc function, 236–239
The clickV function, 238–239
The commandKey function, 249–251
The date function, 381–385
conversion functions for, 388–392
The long date function, 382, 384
The long time function, 386, 391
The long version function, 457–458
The message box. *See* Message box
The mouse function, 239–241
The mouseClick function, 241–243, 327
The mouseH function, 232–233
The mouseLoc function, 234–236
The mouseV function, 233–234
The number of function, 136–137, 311, 360–362
The optionKey function, 249–251
The param of function, 262, 446–448
The paramcount function, 449–450
The params function, 446–447, 647
The random of function, 52, 177–179, 405
The result function, 455–456
with XCMDs, 615
The screenRect function, 582–583, 645
The seconds function, 392–396
The selectedChunk function, 154–155, 370–371
The selectedField function, 154–155, 370
The selectedLine function, 154–155, 370–371
The selectedText function, 154, 370
The selection, 648
as container, 152–155, 369–373
and locked fields, 478
The shiftKey function, 249–251
The short date function, 382
The short time function, 386
The sound function, 484–485
The target function, 86–87, 454
The ticks function, 392–396
The time function, 385–387
conversion functions for, 388–392

The tool function, simulation, of, 500–502
The value function, 155
The version function, 392, 456–458, 566
Then *See* if then control statements
This descriptor, 87–88, 95
Three constant, 134, 167
Ticks, 76
  function for, 392–396
  and wait command, 121–122
Time and time() function, 385–387
  conversion of, 388–392
  date information with, 387–388
  elapsed, 392–396
  European, 384–385
Times with repeat loops, 310
Title bars, 11, 60
To, 50
Tool Palette, 22, 500
  selection of tools from, 118–119
  showing of, 550
Tools and toolbox, 9, 21
  book for, 622–623
  switching between, 50
  with XCMDs and XFCNs, 625
Tools menu, 11
Top property, 510, 584–585
TopLeft property, 510, 584–585
Trace Edges command, 545
Transparency and transparent command, 65
  and select command, 548
Trigonometric functions, 412–419
Triplets, musical, 473
True constant, 135, 168, 241, 272
  See also Boolean expressions
Trunc() function, 182–183, 402–403
Two constant, 134, 167
Type command, 264–265
  simulation of, 499–500
  with text tool, 532
Typing level, 32, 561

U

Underscore character (_) in names, 142
Undo command, 498, 549
Unlock screen command, 572, 645
Until
  with repeat structures, 320–324
  with wait, 334–335
Up constant, 135, 239, 249
  with arrowKey message, 262
  with visual effects, 112
Upgrades of HyperCard, 458
Uppercase letters, 38
  and ASCII codes, 352–353
  with comparisons, 282
  with testing, 365
User assistance, 90
User-defined commands, 441–442
  hierarchy of, 450–451
  making of, 443–444
  parameters with, 444–450
User-defined functions, 430
  design of, 431–434
  override by, 429
  parameters with, 432–438, 440–441
  returning values with, 438–440
User interface
  keyboard functions for, 251–259
  scripts for, 336–338
User levels, 7, 20–21, 31–33, 490, 559
User Preferences card, 32
  and global properties, 559–565
UserLevel property, 560–562
UserModify property, 575–577
Utilities for HyperCard, 23–24

V

Values, 129
  sources of, 132–136
Variable number of parameters, function for, 449–450
Variables, 129–132
  addition of, 134
  as containers, 141–152
  global, 143–146, 157, 624
  initialization of, 404–405
  local, 146–149
  names for, 142–143
  with XCMDs and XFCNs, 624, 627–628
Venetian blinds visual effect, 110–111
Verhulst's Law, 191–194

`Version()` function, 392, 456–458
Versions of HyperCard, 31, 566, 645–648
Vertical position of mouse, 233–234, 238–239
`Very fast` modifier, 107, 109
`Very slowly` modifier, 107, 109
`Visible` property, 505–506, 508–509
  for windows, 583–584
`Visual` command and visual effects, 106–116
  and Macintosh II, 108, 524
Voice parameters, 466–467
  with `play`, 103–104
Volume of sound, 465

W

`Wait` command, 76, 121–122, 334–336
  for pauses, 310–311
  `wait until`, 334–335
  `wait while`, 335–336
`WaitNextEvent` command with XCMDs and XFCNs, 626
Well, The, 24
`While` with `wait`, 335–336
`White` modifier, 107, 109
White space with words, 356
`WideMargins` property, 507
`Width` property, 510, 584
WildCard, 6
Windoids, 21–22, 33
Windows, 10–11, 577
  `location` property for, 578–581
  `rectangle` property for, 581–586
  `visible` property for, 583–584
Winkler, Dan, 8
`Wipe` visual effect, 111–114
Word 3 files, reading of, 604
Word chunks, 355
Word counting routines, 319–320
  with buttons, 362–363
Wrapping of text, 316
`Write` command and writing to files, 351, 593, 595–596, 599–600
Write-protection of stacks, 576–577, 645

X

XCmdBlock data structure, 628–629
XCmdGlue.inc callback, 618
XCMDs and XFCNs, 615–617
  building of, 624–628
  callbacks for, 618–622
  ChangeCase example, 630–643
  data structure for, 628–629
  DoList, 125
  resources for, 622–623

Y

Years, conversion functions for, 389

Z

Zero, division by, 172–173
`Zero` constant, 134, 167
Zero-terminated character arrays, 616
`ZeroBytes` callback, 619
Zeroes in formatting, 189–190
`ZeroToPas` callback, 619
Zones of stability in chaos theory, 191, 193
`Zoom` visual effect, 114–116

The Waite Group

100 Shoreline Highway, Suite 285 | Mill Valley, CA 94941 | (415) 331-0575
Compuserve: 74146,3515 | usernet: bplabs!well!mitch | AppleLink: D2097

Dear Reader:

Thank you for considering the purchase of our book. Readers have come to know products from **The Waite Group** for the care and quality we put into them. Let me tell you a little about our group and how we make our books.

It started in 1976 when I could not find a computer book that really taught me anything. The books that were available talked down to people, lacked illustrations and examples, were poorly laid out, and were written as if you already understood all the terminology. So I set out to write a good book about microcomputers. This was to be a special book—very graphic, with a friendly and casual style, and filled with examples. The result was an instant best-seller.

Over the years, I developed this approach into a "formula" (nothing really secret here, just a lot of hard work—I am a crazy man about technical accuracy and high-quality illustrations). I began to find writers who wanted to write books in this way. This led to co-authoring and then to multiple-author books and many more titles (over seventy titles currently on the market). As The Waite Group author base grew, I trained a group of editors to manage our products. We now have a team devoted to putting together the best possible book package and maintaining the high standard of our existing books.

We greatly appreciate and use any advice our readers send us (and you send us a lot). We have discovered that our readers are detail nuts: you want indexes that really work, tables of contents that dig deeply into the subject, illustrations, tons of examples, reference cards, and more.

I think you will find that **The Waite Group's HyperTalk Bible** is a good example of The Waite Group formula for a computer book. You'll find facts, tips, boxes, quizzes, projects, icons, warnings, and much more. If you like this book, you may want to take a look at its companion title, **The Waite Group's Tricks of the HyperTalk Masters**. That book takes off into the more advanced topics of HyperTalk, revealing little-known, powerful tricks.

If you'd like to extend your programming experience into the fast and powerful C language, you'll want to study our book **C Primer Plus**. This book teaches you generic ANSI C and prepares you for the next step of learning the complex Macintosh Toolbox. There are also a number of good books in the Hayden Macintosh Library you should check out. Watch for future Waite Group books in the Macintosh Library. A list of all our titles can be found in the back of this book. In fact, let us know what you want and we'll try to write about it.

Thanks again for considering the purchase of this title. If you care to tell me anything you like (or don't like) about the book, please write or send email to the addresses on this letterhead.

Sincerely,

Mitchell Waite

Mitchell Waite
The Waite Group

Dear Reader:

If you enjoyed this book, you may be interested in these additional subjects and titles from The Waite Group and Howard W. Sams & Company. Reader level is as follows: ★ = introductory, ★★ = intermediate, ★★★ = advanced. You can order these books by calling 800-428-SAMS.

Level	*Title*	*Catalog #*	*Price*	
	C and C++ Programming Language			
Tutorial, UNIX & ANSI				
★	C Primer Plus, Revised Edition, Waite, Prata, & Martin	22582	$24.95	
★★	C++ Programming, Berry	22619	$24.95	NEW
★★★	Advanced C Primer ++, Prata	22486	$24.95	
Tutorial, Product Specific				
★	Microsoft C Programming for the PC, Revised Edition, Lafore	22661	$24.95	NEW
★	Turbo C Programming for the PC, Revised Edition, Lafore	22660	$22.95	NEW
★★	Inside the Amiga with C, Second Edition, Berry	22625	$24.95	NEW
Reference, Product Specific				
★★	Microsoft C Bible, Barkakati	22620	$24.95	NEW
★★	Quick C Bible, Barkakati	22632	$24.95	NEW
★★	Turbo C. Bible, Barkakati	22631	$24.95	NEW
★★	Essential Guide to ANSI C, Barkakati	22673	$7.95	NEW
★★	Essential Guide to Turbo C, Barkakati	22675	$7.95	NEW
★★	Essential Guide to Microsoft C, Barkakati	22674	$7.95	NEW
	DOS and OS/2 Operating System			
Tutorial, General Users				
★	Discovering MS-DOS, O'Day	22407	$19.95	
★	Understanding MS-DOS, O'Day & Angermeyer	27067	$17.95	
Tutorial/Reference, General Users				
★★	MS-DOS Bible, Second Edition, Simrin	22617	$22.95	NEW
Tutorial/Reference, Power Users				
★★	Tricks of the MS-DOS Masters, Angermeyer & Jaeger	22525	$24.95	
Tutorial, Programmers				
★★	MS-DOS Papers, Edited by The Waite Group	22594	$26.95	NEW
★★	OS/2 Programmer's Reference, Dror	22645	$24.95	NEW
★★★	MS-DOS Developer's Guide, Revised Edition, Angermeyer, Jaeger, et al.	22630	$24.95	NEW
	UNIX Operating System			
Tutorial, General Users				
★	UNIX Primer Plus, Waite, Prata, & Martin	22028	$22.95	
★	UNIX System V Primer, Revised Edition, Waite, Prata, & Martin	22570	$22.95	
★★	UNIX System V Bible, Prata and Martin	22562	$24.95	
★★	UNIX Communications, Henderson, Anderson, Costales	22511	$24.95	
★★	UNIX Papers, Edited by Mitchell Waite	22570	$26.95	
Tutorial/Reference, Power Users and Programmers				
★★	Tricks of the UNIX Masters, Sage	22449	$24.95	
★★★	Advanced UNIX—A Programmer's Guide, Prata	22403	$24.95	
	Macintosh			
Tutorial, General Users				
★	HyperTalk Bible, The Waite Group	48430	$24.95	NEW
Tutorial/Reference, Power Users and Programmers				
★★	Tricks of the HyperTalk Masters, Edited by The Waite Group	48431	$24.95	NEW

If you liked Tricks of the HyperTalk Masters...

... you can continue your journey into HyperTalk with Macintosh Hands On magazine. Each month, you'll find Mitchell Waite's column, HyperTalk Tricks. Mitch shows you how to extend HyperTalk, reveals great tips, secrets, and tools that improve your scripting and stacks, like the ones in this book. You'll find Macintosh Hands On magazine a source of inspiration and a great collection of articles, stacks, and creative ideas for HyperCard scripting.

MACINTOSH
Hands On™
The Personal Macintosh Magazine

Get a new HyperCard stack project EVERY MONTH... for home finance, investing, art, education, music, personal productivity, and sheer fun. Each project has clear listings and instructions for entering them. No other magazine offers this kind of value.

Boost your productivity... with secrets and techniques that the experts use for Microsoft Excel, Word, PageMaker, XPress, SuperPaint, and other major programs.

Learn the basics... with Your First Stack, a regularly featured starter project that's short, sweet, and easy to follow. You'll be surprised how easy it is.

Sharpen your HyperCard scripting... with advanced techniques for graphing, using stacks to create themselves, making stacks that analyze other stacks, and much more.

And Macintosh Hands On's stack listings are easier to read and use, by far, than those of any other magazine.

We guarantee that you'll like Macintosh Hands On or we'll refund your money for all unmailed issues.

You SAVE 30% off the cover price. PLUS you get a special FREE Disk!

YES! I want to get Macintosh Hands On and save 30%! In addition, I want the special disk — Secrets of the Mac Experts — a handy HyperCard stack with dozens of tips and tricks on HyperCard, Word, Excel, PageMaker, and other best sellers (free with my paid subscription). Start my Hands On subscription immediately.

U.S. Rate: $24.95 for 12 issues Payment enclosed: ☐ Check/Money Order ☐ VISA ☐ MasterCard

Name _____ Credit Cd # _____ Expires _____

Address _____ Signature _____

City _____ State _____ Zip _____ Tel. # _____

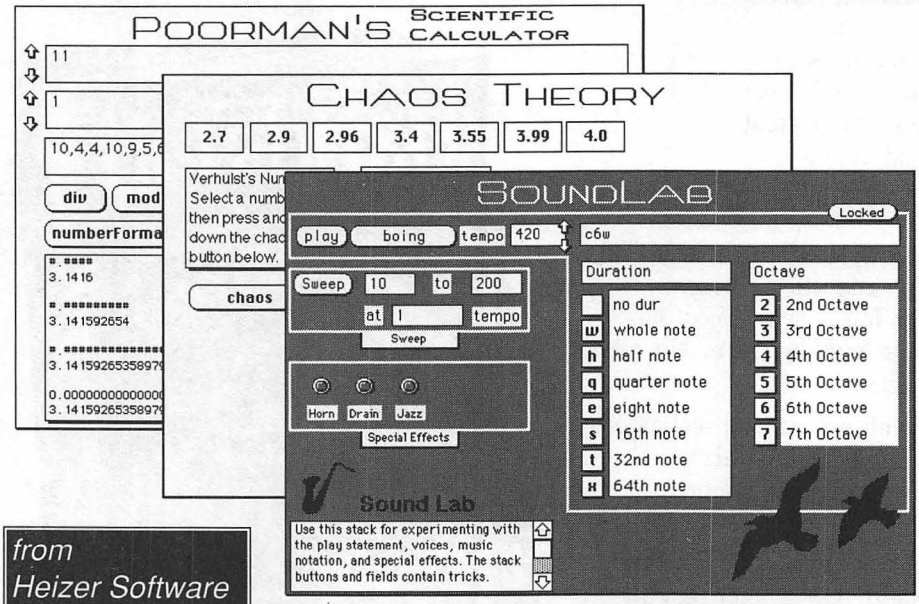
Foreign rates: Canada surface $32.95. Overseas surface $37.95. All payments must be in U.S. funds drawn on a U.S. bank. TRAD

Mac Hands On, 52 Domino Drive, Concord MA 01742 or call (508) 371-1660 with your MasterCard or Visa.

Make your scripting faster and easier with...

HyperTalk Bible: The Stack

All the fonts, tools, scripts, sounds, graphics, and custom artwork you need to begin immediately using the ideas in this book. The disks contain the SoundLab, Chaos Theory, and the Poorman's Scientific Calculator stacks, as well as all the cards and scripts contained in the book.



from Heizer Software

SPECIAL BONUS

As a special bonus to readers of *HyperTalk Bible*, we are offering a free copy of Poly-Buttons, the amazing script that allows you to create, cut and paste multisided buttons! With PolyButtons you can cover shapes that would normally take many rectangular buttons. A $19.95 value FREE when you order *HyperTalk Bible: The Stack*.

If you want to avoid typing the scripts in this book, as well as to obtain the graphic screens, fonts, external commands, and other custom resources used in this book, you want *HyperTalk Bible: The Stack* from Heizer Software's Stack Exchange. The stack is available for $19.95 (California residents, please add sales tax) plus $3.00 shipping and handling. To order, just fill out the coupon below or call Toll Free: **800-888-7667, weekdays 9am - 5 pm, Pacific time**. When you call, have your credit card and reference product code 30243 available.

The StackExchange is a clearinghouse for programs written in Apple's HyperCard. If you are new to HyperCard, you'll find our tutorials and ready-to-go tools will help you become productive quickly. If you are an advanced user, you'll find many useful utilities that help to keep development time to a minimum. The Heizer Software catalog contains hundreds of templates, tutorials, and databases for Microsoft Excel and Works.

☐ Yes, I want the Macintosh disk containing *HyperTalk Bible: The Stack* (#30243), as well as my free *PolyButtons* stack, for $19.95. And don't forget a free copy of the Heizer Software catalog!

☐ No, I'll type it all manually and figure out my own way to do the graphics, but do send me that free Heizer Software catalog.

Name: _____

Address: _____

_____

City, State, Zip: _____

Phone: _____

☐ US$ Check enclosed (payable to Heizer Software) # _____

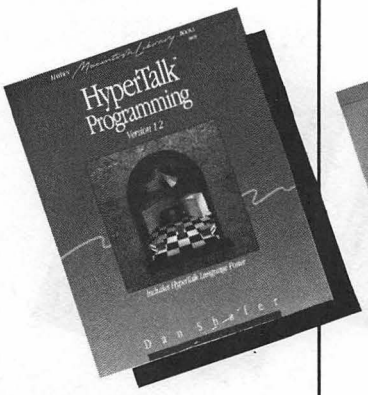
☐ VISA ☐ MasterCard ☐ American Express

Name as it appears on card _____ Expiration Date _____

Card No. _____

Signature (we cannot process credit card orders without your signature) _____

Heizer Software • P.O. Box 232019 • Pleasant Hill, CA 94523 • Order Line: 800-888-7667, Main Offices: 415-943-7667, FAX: 415-943-6882



HyperTalk™ Programming Version 1.2, Revised Edition

Dan Shafer

This comprehensive tutorial covers every feature of HyperTalk Version 1.2 from basic theory to practical examples. Two complete scripts for applications are provided including one which enables "semi-automatic" programming. Details on how to extend HyperTalk with other programming languages are included as is an explanation outlining how to use graphics, sound, and communications. A special wall poster provides a quick reference to HyperTalk commands and functions.

Topics covered include:

- Object-Oriented Programming Concepts
- System Messages
- Keyboard, Mouse, and File I/O
- Control Structures and Logical Ops
- Card and Stack Control Methods
- Text and Data Management
- Dialogs
- Managing Menus and Tools
- Graphics and Visual Effects
- Sound and Music Basics
- Math Operators and Functions
- Action-Taking, Property-Related, and Communications Commands
- Script-Related and Miscellaneous Commands
- Tips, Traps, and Techniques
- Extending HyperCard with Resources, Icons, and External
- Tools and Programming Aids
- HyperQuiz and Semi-Automatic Programming Script
- Appendices: HyperTalk Vocabulary, ASCII Chart, HyperCard Resources

600 Pages, 7¾ x 9¼, Softbound
ISBN: 0-672-48439-0
No. 48439, $24.95

IBM® PC and Macintosh® Networking

Stephen L. Michel

IBM PC and Macintosh owners and users who want to combine the power of their machines will welcome this complete resource for networking the IBM PC and the Macintosh using TOPS and AppleShare.

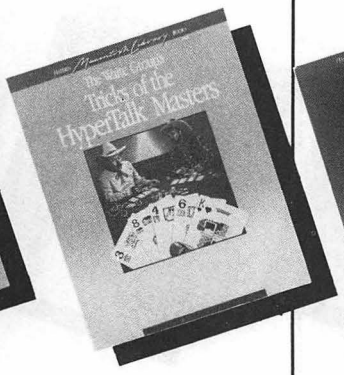
This book details the specifics of using the Macintosh and the IBM PC on the same network, including transferring files, sharing printers, transporting data from IBM software to Mac and vice versa, and mixing word processing and spreadsheet programs.

Full of networking details, this thorough coverage of TOPS software (one of *PC Magazine's* "The Best of 1986" products) details how to create useful files and share printers and external disk drives.

Topics covered include:

- How the Macintosh and PC Really Differ
- TOPS
- AppleShare
- Coexistence
- Managing the Network
- Appendices: Glossary, ASCII Character Sets, Using PostScript Printers

328 Pages, 7¾ x 9¼, Softbound
ISBN: 0-672-48405-6
No. 48405, $21.95



Tricks of the HyperTalk™ Masters

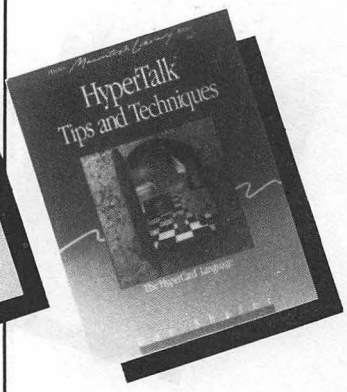
Edited by The Waite Group

Written for HyperTalk "scripters," this advanced guide brings together tips, tricks, and tools from numerous experts. These HyperCard developers share their advice and show the reader how to do the "impossible." Dozens of text field, button, and menu-related scripts, graphics and sound techniques, tool and utility scripts, tips on measuring and increasing stack performance, and handlers for reporting the contents of the stack objects are discussed in a clear, concise style.

Topics covered include:

- Techniques for Stack Development
- ScrollJump
- Dueling Scroll
- Quick PopUp Fields
- Menu Fields
- PolyButtons
- Animated Icons, Cursors, and Fonts
- The Art of Visual Effects
- Synching Sight and Sound
- Developer's Tools
- The Quiz Maker and Hyper-Animals
- A Grab Bag of Utilities
- Chauffeur
- Benchmarking HyperTalk, Large Stacks
- Extending HyperTalk with Externals

616 Pages, 7¾ x 9¼, Softbound
ISBN: 0-672-48431-5
No. 48431, $24.95



HyperTalk™ Tips and Techniques

Dan Shafer

Written for programmers and developers, this book is a collection of more than 100 helpful pieces of information about HyperTalk, the programming language built into Apple® 's HyperCard™. It offers readers with some experience in HyperTalk programming a chance to learn the ins and outs of programming from one of the best-known and widely recognized HyperTalk scripting experts.

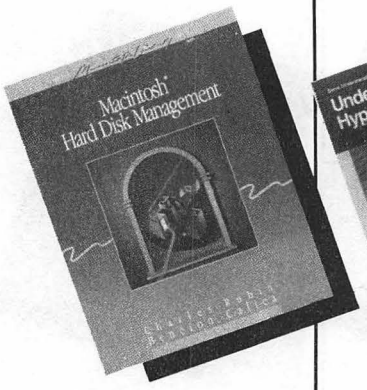
Solutions to dozens of bugs, deficiencies, and pitfalls lying in wait for the unsuspecting HyperTalk programmer are documented as are suggestions for handling some of the most often needed HyperTalk tasks. The book provides special shortcuts, speed-ups, and enhancements and a wealth of additional information that isn't available from any other source.

Topics covered include:

- Creating an Invisible Cursor
- Building an Index of Stack Contents Automatically
- Checking a Field's Content for Data Type
- Multiword and Multifield Signs
- Protecting a Stack and Script from Mis-use
- How to Construct HyperText Applications in HyperCard
- Dealing with HyperCard's Limits and Performance Issues

300 Pages, 7¾ x 9¼, Softbound
ISBN: 0-672-48427-7
No. 48427, $21.95

Visit your local book retailer or call
800-428-SAMS.



Macintosh® Hard Disk Management

Charles A. Rubin and Bencion J. Calica

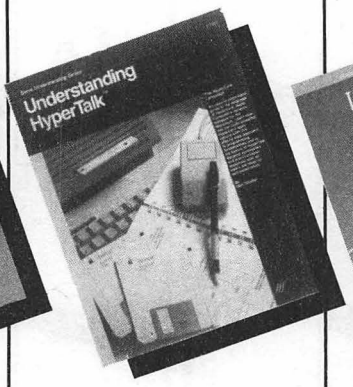
This is the ideal companion book for all Macintosh owners who have a hard disk or are considering the purchase of one.

Readers will discover how the disk works, as well as pick up important information on how to recover files, rebuild the desktop, replace files, and install fonts and desk accessories.

Topics covered include:

- Hardware
- The System Folder Files
- The Finder
- Fonts, DAs, and the Font/DA Mover
- Organizing Hard Disk Files
- Sharing Files with a Hard Disk
- Using File or Disk Copying Programs
- Printing from a Hard Disk
- Backing Up a Hard Disk
- Disk Optimizing Utilities
- Font and Disk Accessory Extenders
- Fkeys, Inits, and Chooser Resources
- Finder Alternatives
- Preventive Measures
- Troubleshooting and Repairs
- Appendices: Glossary of Terms, List of Products

300 Pages, 7¾ x 9¼, Softbound
ISBN: 0-672-48403-X
No. 48403, $19.95



Understanding HyperTalk™

Dan Shafer

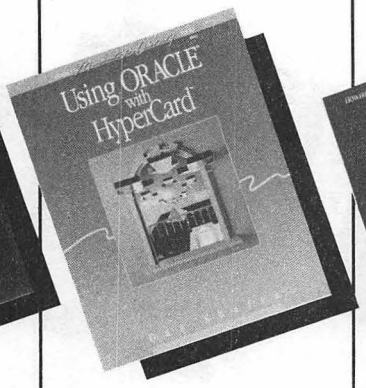
Understanding HyperTalk brings the power and fascination of programming in HyperTalk to those Macintosh® owners who want to customize their environment with Apple® 's HyperCard™.

Written by the author of the best-selling *HyperTalk Programming*, this book will be most useful to people who are deciding whether to buy HyperCard and to people who want to teach themselves or others HyperCard programming and stacks.

Topics covered include:

- Programming Basics
- Object-Oriented Programming Ideas
- HyperCard Refresher
- HyperTalk Building Blocks
- System Messages
- Input/Output
- Loops and Conditional Processing
- Navigational Commands
- Data Management Commands
- User Interface Commands
- Graphics and Visual Effects
- Sound and Music
- Math Functions and Operators
- Action-Taking Commands
- Property
- Interface to the Outside World
- Stack Design Considerations

300 Pages, 7 x 9, Softbound
ISBN: 0-672-27283-0
No. 27283, $17.95



Using ORACLE® with HyperCard®

Dan Shafer

Macintosh users will welcome this tutorial on ORACLE which shows how to use HyperCard to design and build front ends to ORACLE databases. The book includes several case studies which take the reader step-by-step through an application, showing how to program in SQL; how to design, build, and program HyperCard stacks for database access; and how to use Hyper*SQL. It includes full source code for three typical interface examples that become templates for future applications.

Topics covered include:

- ORACLE Database Management Concepts
- An Overview of SQL
- Design Considerations for HyperCard
- Designing HyperCard Stacks for Database Use
- Building a HyperCard Stack
- HyperTalk Programming Basics
- The HyperTalk Language for ORACLE Access
- Hyper*SQL Language Extensions
- Connectivity Concerns
- The Basic Tasks in ORACLE for HyperCard
- Advanced Programming and Usage
- Local Database, Single Stack
- Remote Database, Single Table
- Complex Example
- Appendices: SQL Command Reference, HyperTalk Language Reference, Error Messages and Codes

350 Pages, 7¾ x 9¼, Softbound
ISBN: 0-672-48443-9
No. 48443, $24.95

680x0 Programming by Example

Stan Kelly-Bootle

This sequel to the *68000, 68010, 68020 Primer* teaches the finer points of M68000 assembly language with hundreds of practical examples.

For the complete novice there is a detailed guide to the M68000 instruction set and addressing modes, as well as an introduction to assemblers, linkers, and loaders. For the advanced programmer there are new insights and tips for producing fast, tight, bugfree code. Special attention is given to the full 32-bit MC68020 and the latest enhanced MC68030, Motorola's answer to Intel's 80386.

Topics covered include:

- Assembly Language Pros and Cons
- M68000 Programmer's Models
- M68000 Addressing Modes
- M68000 Instruction Sets
- Assembler Basics, Including the Latest Structured Assemblers
- Deja Code—Its Cause and Cure
- String Manipulation
- Binary Coded Arithmetic
- Common Errors and Debugging
- Sorting and Searching
- Complete Source Code for the Popular KERMIT™ File-transfer Protocol

512 Pages, 7½ x 9¾, Softbound
ISBN: 0-672-22544-1
No. 22544, $17.95

Visit your local book retailer or call
800-428-SAMS.

Macintosh® Revealed, Volume One: Unlocking the Toolbox, Second Edition

Stephen Chernicoff

Macintosh Revealed, Volume One is the most successful Macintosh technical programming book commercially published. Now, in addition to covering the 128K and 512K Macintosh, the second edition concentrates on the Macintosh Plus and its expanded memory management, I/O routines, graphics and text handling, and serial communications.

This edition has been completely updated with all the latest Mac features including the new keyboard layouts, character codes for expanded LaserWriter,™ font families and access, ROM resources, and graphic resources. Together with Volume Two, the nearly 500 ROM routines that make up the Macintosh Toolbox are described.

Topics covered include:

- All the Tools You Need
- Putting the Tools to Work
- Thanks for the Memory
- Any Port in a Storm
- Quick on the Draw
- Summoning Your Resources
- Getting Loaded
- Upstanding Characters
- Appendices: Toolbox Summary, Resource Formats, Macintosh Memory Layouts, Key Codes and Character Codes, Error Codes, Summaries of Assembly-Language Variables, and Trap Macros and Trap Words.

590 Pages, 7¾ x 9¼, Softbound
ISBN: 0-672-48400-5
No. 48400, $26.95

Macintosh® Revealed, Volume Two: Programming with the Toolbox, Second Edition

Stephen Chernicoff

Providing an advanced look at the higher-level parts of the Toolbox that implemented the revolutionary Macintosh user interface, this second edition focuses on the Macintosh Plus as well as the 128K and 512K Macintosh.

This new edition of Volume Two includes HFS, 800K double-sided disks, the new Standard File and Disk Initialization Package, window zooming, zoom boxes, scrollable menus, function key routines, and automatic text scrolling.

Software featured in the book is available on disk from the author.

Topics covered include:

- All the Tools You Need
- Keeping Up with Events
- Windows on the World
- What's on the Menu?
- Scissors and Paste
- At the Controls
- Meaningful Dialogs
- Files at Your Fingertips
- Appendices: Toolbox Summary, Resource Formats, Memory Layouts, Key Codes and Character Codes, Error Codes, Assembly-Language Variables, Trap Macros and Trap Words, and MiniEdit Source Listing

696 Pages, 7¾ x 9¼, Softbound
ISBN: 0-672-48401-3
No. 48401, $26.95

Macintosh® Revealed, Volume Three: Mastering the Toolbox

Stephen Chernicoff

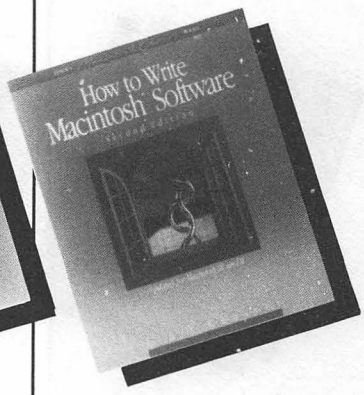
A continuation of the "Macintosh Revealed" series, this third volume focuses on the User Interface Toolbox and how these unique routines can enhance programming.

This Toolbox provides programmers with a uniform set of procedures for manipulating graphic images, text, overlapping windows, pull-down menus, and many other Macintosh features. Through fully worked program examples, the book explores secrets of the Toolbox in great detail and includes such topics as printing and sound generation, the List Manager, the customized Toolbox, desk accessories and Switcher.

Topics covered include:

- Unlocking the Secrets
- Rolling Your Own
- In the Driver's Seat
- Looking Good on Paper
- Sound and Fury
- Accessories after the Fact
- Making a List, Checking It Twice
- Odds and Ends
- Appendices: Toolbox Summary, Resource Formats, Error Codes, Trap Macros and Trap Words, Assembly Language Variables, Program Listings

600 Pages, 7¾ x 9¼, Softbound
ISBN: 0-672-48402-1
No. 48402, $26.95



How to Write Macintosh® Software, Second Edition

Scott Knaster

Written for professional developers and serious hobbyists, this is the best source of information on the intricacies of the Macintosh operating system, and in particular the Memory Manager.

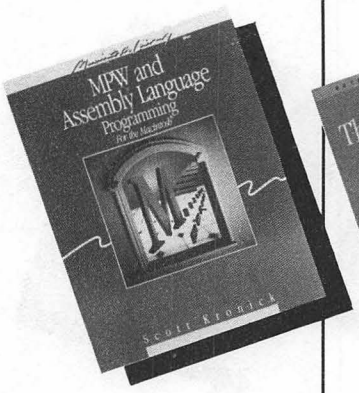
This new edition explains how applications programs on the Macintosh work, how to create and debug professional-quality programs, and how to use C to program the Macintosh. Many new topics, including Macintosh II, Macintosh SE, MultiFinder, Macintosh Programmer's Workshop, and the 68020 and 68030 microprocessors, are included as well as revised and updated information on all its previous topics. Its in-depth discussion of high-quality debugging makes it the preferred reference for programmers and software applications developers.

Topics covered include:

- Getting Started
- Adding Features
- Writing a Program
- Using C
- Loops
- Functions, Subroutines, and Subprograms
- Using Macintosh Features

600 Pages, 7¾ x 9¼, Softbound
ISBN: 0-672-48429-3
No. 48429, $28.95

Visit your local book retailer or call
800-428-SAMS.



MPW and Assembly Language Programming for the Macintosh®

Scott Kronick

This introduction to MPW for programmers is the first to teach Macintosh assembly language. Macintosh Programmer's Workshop (MPW) is the new programming development system for the Macintosh and one of the most sophisticated microcomputer programming development systems in existence.

This book is a clear and carefully written introduction to MPW, assembly language, Pascal, and C. It will help beginning and intermediate programmers write assembly language, Pascal, and C programs using the Macintosh Toolbox.

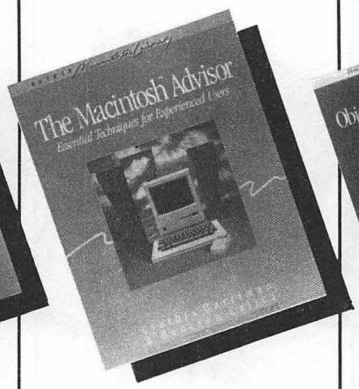
Topics covered include:

The Macintosh Programmer's Workshop
- A Sample Program in Assembly Language
- Fundamental File Commands
- Start Up and Files
- Command Language
- Make and Structured Commands

The Assembly Tutorial
- Slots
- First Lines of Assembly Code
- The ABC's of Blocks of Code
- QuickDraw Inside the Window
- Structured Programming with Blocks
- The Keyboard
- Menus

The MPW and Assembly Dictionaries
- The MPW Shell Command Language
- The 68000 Instruction Set with Directives and Toolbox Traps

352 Pages, 7¾ x 9¼, Softbound
ISBN: 0-672-48409-0
No. 48409, $24.95



The Macintosh® Advisor

Cynthia Harriman and Bencion Calica

Newly updated to include MultiFinder™, this book provides advice on shortcutting some of the Macintosh's elementary procedures to use it more productively.

MultiFinder is the first multitasking operating system for the Mac which allows users to work with multiple applications at the same time. Learn how to use this system upgrade to its fullest extent, while gaining a better understanding of RAM disks, memory management, and other features. The book also examines powerful hardware options such as hard disks, RAM upgrades, and the LaserWriter™ and includes troubleshooting procedures for quickly isolating and solving computer problems.

Topics covered include:

- The Finder: Macintosh's Operating System
- Speeding Applications
- Desk Accessories and FKeys
- Customizing Applications
- Disk Management Utilities
- RAM Upgrades and Hard Disk Drives
- Options for Better Input and Output
- IBM to Mac: Transferring Data
- Troubleshooting
- Appendices: Shopping Lists, Sources, Technical References

320 Pages, 7¾ x 9¼, Softbound
ISBN: 0-8104-6569-8
No. 46569, $19.95

Object-Oriented Programming for the Macintosh®

Kurt J. Schmucker

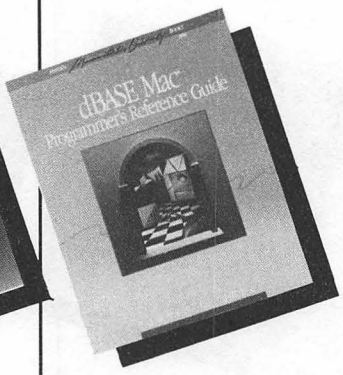
With this book, gain insight into the fundamental object-oriented concepts of objects, classes, instances, message passing and method calls, and into advanced topics like metaclasses and multiple inheritance.

Learn to customize MacApp and avoid programming resizable windows, dialog boxes, and scroll bars from scratch. Investigate QuadWorld and the major Macintosh object-oriented languages.

Topics covered include:

- Why Object-Oriented Programming?
- The Basics of Object-Oriented Programming
- Object Pascal
- Introduction to MacApp
- Mini-QuadWorld—A Small MacApp Application
- The Most Frequently Asked Questions about MacApp
- The Flow of Events in MacApp
- QuadWorld—A Full MacApp Application
- Advanced MacApp Features
- Advanced Concepts in Object-Oriented Programming
- Smalltalk
- Lisa Clascal and the Lisa Toolkit
- An Overview of Other Object-Oriented Languages on Macintosh
- Appendices: The Macintosh User Interface Standard, and Object-Oriented Programming Languages, Suppliers and User Groups

624 Pages, 7¾ x 9¼, Softbound
ISBN: 0-8104-6565-5
No. 46565, $34.95



dBASE Mac® Programmer's Reference Guide

Edward C. Jones

This detailed guide for using and programming in dBASE for the Macintosh® was written especially for those who need to take full advantage of the program's available power.

This book highlights programming applications using modular code to develop complete applications for any purpose. It provides a detailed explanation of how to use the relational powers of the software, as well as how to add and edit records with full error tracking and data verification, system design, and file transfer from DOS.

Current DOS dBASE users will learn how to take advantage of the unique Macintosh graphic interface when they program or use dBASE Mac.

Topics covered include:

- Creating and Modifying Files
- Working with Display Views
- Managing Data
- Working with Related Files
- Building Applications
- Procedural Flow in a dBASE Mac Procedure
- Adding and Modifying Records Under Procedural Control
- User Input and Output
- dBASE Mac for dBASE III Programmers
- Sample/Design Applications
- Appendix: dBASE Mac Commands

300 Pages, 7¾ x 9¼, Softbound
ISBN: 0-672-48416-1
No. 48416, $19.95

Visit your local book retailer or call
800-428-SAMS.

HyperTalk Version 1.2.2 Language Syntax

(continued from inside front cover)

mouseDown, *205*
mouseEnter, *203*
mouseLeave, *203*
mouseStillDown, *205*
mouseUp, *206*
mouseWithin, *207*
newBackground, *202*
newButton, *202*
newCard, *202*
newField, *202*
newStack, *202*
openBackground, *202*
openCard, *202*
openField, *202*
openStack, *202*
quit, *218*
resume, *216*
returnInField (1.2), *210*
returnKey, *210*
startUp, *43*
suspend, *216*
tabKey, *211*

6. Miscellaneous

--	comment - chars on its right up to carriage return are ignored., *43*
"done"	literal used for: wait until the sound is "done", *485*
the	preceeds function name to indicate 2nd form of functions, *440*
it	local variable that is default destination for get, ask, answer, read and convert, *149*

External Commands and Functions

flash n	inverts screen and returns to normal, *616*

Error Dialog

cantUnderstand, *170*

Built-in Voices

"harpsichord", *466*
"boing", *466*
"silence", *466*

Prepositions

after, *46*
into, *46*

Adjectives

abbr
long, *88*
short, *88*

Ordinals

any, *95*
eighth
fifth
fourth
last, *95*
mid[dle], *95*
ninth
second
seventh
sixth
tenth
third

Object Modifiers

next, *95*
prev[ious], *95*
this, *85*

Chunk Type

char[acter], *355*
items, *355*
line, *355*
word, *355*
words, *355*

Synonyms

bkgnd	background, *77*
bkgnds	backgrounds, *77*
btn	button, *62*
cd	card, *59*
loc	location, *504*
msg [box]	message box, *45*
mid	middle, *95*
poly	polygon, *539*
rect	rectangle, *504*
reg	regular

Synonyms (1.2)

bg	background
bgs	backgrounds
btns	buttons
cd	card
cds	cards
fld	field
flds	fields
grey	gray
pict	picture, *551*
sec	secs or seconds
second	secs or seconds
tick	ticks, *392*

7. Operators

Arithmetic Operators

()	grouping for evaluation, *184*
-	minus: changes sign of a num or subtracts right number from left, *165*
+	plus: adds two numbers, *164*
*	times: multiplies two numbers, *165*
/	divides two numbers, with fractional remainders, *165*
div	divide and truncate: returns the truncated quotient of two numbers, *173*
mod	modulo: divides but returns remainder of two numbers, *174*
^	exponent: raises number on left to power on its right, *174*

Logic Operators

not	boolean not logical operator, *236*
and	boolean and logical operator, *283*
or	boolean or logical operator, *284*
=	logical equals: (text and numbers), *274*
is	same as = , logical equals: (text and numbers), *274*
<>	not equal operator (text and numbers), *277*
≠	Same as < >, *278*
<	less than, *277*
>	greater than, *277*
<=	less than or equal, *277*
≤	same as <= (Option-comma), *278*
>=	greater than or equal to, *277*
≥	same as = (Option-period), *279*

Text Logical Operators

contains	contains true if source string on left is in string on right, *364*
is in	opposite of contains, true if string on right is in left, *364*
is not in	opposite of is in, *277*

String Operators

&	concatenates two strings, *347*
&&	concatenates two strings with a space between them, *347*

8. Version 1.2 Shortcuts

Command-Tab (Cmd-Tab)

Cmd-Tab chooses Browse tool.
(1.2) Only:
Cmd-Tab (2x): chooses Button tool
Cmd-Tab (3x): chooses Field tool
(rapid succession = 30 ticks)

Command-Option (Cmd-Option)

In Browse tool:
Cmd-Option shows outlines of all visible buttons. Click outline to edit script.
In Button tool:
Cmd-Option shows outlines of all visible and hidden buttons, but click to edit only works on visible btns.

Shift Command-Option (Shft-Cmd-Option)

In Browse tool:

HyperTalk Version 1.2.2 Language Syntax

Shift Command-Option (continued)

Shft-Cmd-Option outlines all visible fields. Click on outline to edit script.

In Field tool:

Shft-Cmd-Option outlines all visible and hidden fields, but click to edit only works on visible flds.

Other Command-Option Keys

Cmd-Option-C edits script of card
Cmd-Option-B edits script of background
Cmd-Option-S edits script of stack

9. Properties

Global Properties

blindTyping, *563*
cursor, *566*
dragSpeed, *567*
editBkgnd, *569*
language, *570*
lockScreen, *573*
lockRecent, *572*
lockMessages, *573*
numberFormat, *188*
powerKeys, *563*
textArrows, *562*
userLevel, *560*

Window Properties

loc[ation], *578*
rect[angle], *581*
visible, *583*

Painting Properties

brush, *527*
centered, *535*
filled, *524*
grid, *540*
lineSize, *526*
multiple, *537*
multiSpace, *539*
pattern, *550*
polysides, *539*
textAlign, *525*
textFont, *525*
textHeight, *525*
textSize, *525*
textStyle, *525*

Stack Properties

freeSize, *504*
name, *137*
script, *35*
size, *504*

Card and Background Properties

id, *82*
name, *137*
number, *82*
script, *35*

Field Properties

autoTab (1.2)
id, *82*
loc[ation], *505*
lockText, *63*
name, *505*
number, *505*
rect[angle], *505*
script, *505*
scroll, *505*
showLines, *505*
style, *505*
textAlign, *505*
textFont, *505*
textHeight, *505*
textSize, *505*
textStyle, *505*
visible, *505*
wideMargins, *505*

Button Properties

autoHilite, *505*
hilite, *505*
icon, *567*
id, *505*
loc[ation], *505*
name, *505*
number, *505*
rect[angle], *505*
script, *505*
showName, *505*
style, *505*
textAlign, *505*
textFont, *505*
textHeight, *505*
textSize, *505*
textStyle, *505*
visible, *508*

Modification Properties (1.2)

CantDelete, *645*
CantModify, *645*
UserModify

Misc Properties (1.2)

showPict, *551*
the left of *partOrWindow*, *584*
the top of *partOrWindow*, *584*
the right of *partOrWindow*, *584*
the bottom of *partOrWindow*, *584*
the topLeft of *partOrWindow*, *584*
the bottomRight of *partOrWindow*, *584*
the width of *partOrWindow*, *584*
the height of *partOrWindow*, *584*
point is [not] within *rectangle*

10. Tools

browse, *528*
brush, *527*
bucket, *531*
button, *143*
curve, *539*
eraser, *548*
field, *594*
lasso, *545*
line, *526*
oval, *536*
pencil, *525*
poly[gon], *539*
rect[angle], *534*
reg[ular] poly[gon], *539*
round rect[angle], *535*
select, *541*
spray [can], *530*
text, *532*

11. Containers

field, *138*
it, *149*
message [box], *155*
selection, *152*

12. Repeat

repeat forever, *33*

```
repeat [forever]
      put "hello" into msg box
end repeat
```

repeat for, *310*

```
repeat [for] number [times]
      put "hello" into msg box
end repeat
```

repeat until, *320*

```
repeat until condition
      put "hello" into msg box
end repeat
```

repeat while, *324*

```
repeat while condition
      put "hello" into msg box
end repeat
```

repeat with, *312*

```
repeat with var = start to finish
      put "hello" into msg box
end repeat

repeat with var = start down to
   finish
      put "hello" into msg box
end repeat
```

13. If

if *condition* then *statement* [else *statement*], *263*
if *condition*
then *statement*
else *statement*, *266*
if *condition* then
      statementList
[else
      statementList]
end if, *269*
if *condition* then
      statementList
else *statementList*, *269*

If no page number is listed, the item is not included in this book

HAYDEN *Macintosh Library* BOOKS

Available on Disk

The Waite Group's HyperTalk™ Bible

Covers HyperCard™ International Version 1.2.2

Become a sage in HyperTalk scripting with this comprehensive, up-to-date tutorial. Written for beginning and intermediate HyperCard users who want to write professional HyperTalk scripts, *The Waite Group's HyperTalk Bible* teaches all versions of HyperTalk, including the international version.

Step-by-step lessons, exercises, and quizzes make this book suited for classroom and corporate instruction as well as self-teaching. "Quick View" summaries introduce each chapter to help you find reference material fast. Hundreds of examples show the entire HyperTalk syntax in detail and demonstrate professional approaches to stack building. Icons identify warnings, tips, tricks, limits, time-savers, and other bits of wisdom.

Learn rare HyperTalk secrets and undocumented features. Get a complete sound lab for experimenting with sound effects and a chaos theory lab for learning math. Study advanced topics like message-passing, recursion, object limits, script optimization and generalization, creative workarounds, and power-scripting. Learn about writing XCMDs from Apple authority Steve Maller.

The authors provide information on ordering a companion stack, with all stacks, scripts, graphics, sounds, fonts, and resources covered in *The Waite Group's HyperTalk Bible.* Use the stack and this book to make your scripting more organized, complete, and powerful.

About the Authors

The Waite Group is a developer of computer, science, and technology books. Acknowledged as a leader in its field, The Waite Group creates book ideas, finds authors, and provides development support throughout the book cycle, including editing, reviewing, testing, and production control for each title. The Waite Group has produced over 70 titles, including such best-sellers as C Primer Plus, Microsoft C Bible, MS-DOS Bible, Tricks of the MS-DOS Masters, and UNIX System V Primer. The Waite Group produces 15 to 20 new computer books each year and has yearly sales of a half-million books. Mitchell Waite, president of The Waite Group, wrote his first computer book in 1976. The Waite Group can be reached at 100 Shoreline Highway, Building A, Suite 285, Mill Valley, CA 94941, 415/331-0575.

ISBN 0-672-48430-7

HAYDEN BOOKS
A Division of Howard W. Sams & Company
4300 West 62nd Street
Indianapolis, Indiana 46268
$24.95 US/48430

P0040 BK:MACINTOSH HYPERTALK BIBLE Price 16.99 08