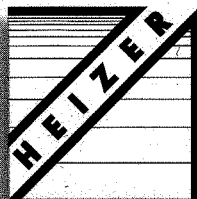


Compilet!

The XCMD Development System



CompileIt!

The XCMD Development System

User Manual

For Technical Support Call

510-943-7667

Monday-Friday, 9 am - 5 pm Pacific time



Heizer Software

P.O. Box 232019 • Pleasant Hill, CA 94523 • 510-943-7667

CompileIt! User Manual ©1990-94 Heizer Software.
All Rights Reserved.

Rev. 5/95

Copyright Notice

You are permitted, even encouraged, to make one backup copy of the enclosed programs. Beyond that is piracy and illegal.

The software (computer programs) you purchased are copyrighted by the author with all rights reserved. Under the copyright laws, the programs may not be copied, in whole or part, without the written consent of the copyright holder, except in the normal use of the software or to make a backup copy. This exception does not allow copies to be made for others, whether or not sold, but the material purchased (together with all backup copies) may be sold, given, or loaned to another party. Under the law, copying includes translating into another language or format. You may use the software on any computer owned by you, but extra copies cannot be made for this purpose. If you have several computers requiring the use of this software, we are prepared to discuss a multi-use or site license with you.

CompileIt! ©1989-1994 Tom Pittman. All Rights Reserved.

DebugIt! ©1991-1994 Tom Pittman. All Rights Reserved.

CompileIt! User Manual ©1990-94 Heizer Software. All Rights Reserved. No part of this document and the software product that it documents may be photocopied, reproduced, or translated to another language without the express, written consent of the copyright holders.

The information contained in this document is subject to change without notice. Heizer Software makes no warranty of any kind with regard to this written material. Heizer Software shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this manual.

CompileIt! and **DebugIt!** are trademarks of Heizer Software. **HyperTalk**, **HyperCard**, **Macintosh**, **Inside Macintosh**, **SANE**, **MPW**, **MacsBug**, and **StackWare** are trademarks or registered trademarks of Apple Computer, Inc. **SuperCard** and **SuperTalk** are registered trademarks of Silicon Beach Software, Inc. **TMON** is a registered trademark of Icom Simulations, Inc. **America Online** is a registered trademark of Quantum Computer Corp. **Compuserve** is a registered trademark of Compuserve, Inc. All other brand or product names are trademarks or registered trademarks of their respective holders.

CREDITS

Project Team

Tom Pittman	Developer
Alan Pabst	Product Manager
Gerald Schmalzried	Software Engineer
Steve Michel	Documentation Review
Ray Heizer	Publisher

Acknowledgements

Extra special thanks to: Mark Thowe, Jerry Daniels, Andrew Meit, and Bryan McCormick for suggestions, killer beta testing, and encouragement. Also to Steve Michel for writing the CompileIt! 1.5 manual which served as a basis for this manual.

Thanks also to all our beta testers, especially: J.P. Adams, James Beldock, Brian Benison, Kevin Calhoun, Jack Carr, Peter Cleaveland, Jeff Close, Keith DeLong, Brad Doster, Danny Goodman, Tom Gruman, Mark Hanrek, David Johnson, Greg Kearney, John Kindschi, Binky Melnik, John Miskimins, Scott Neufeld, Paul Pearson, George Pytlik, Dan Shafer, David Smith, Robertson Reed Smith, Gary Stanoulis, Michel Swain, David Veeneman, Christopher Watson, Dean Wette, Robert Williamson, Roger Wood.

And also, thanks to our customers. Your suggestions and support are greatly appreciated.



TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION	9
What Is CompileIt!	10
Four Reasons for Using CompileIt!	10
How to Use This Manual	10
What Is a Compiler?	11
What Are Externals?	12
Who Should Use CompileIt!	13
What CompileIt! Does and Does Not Do	13
CHAPTER 2 COMPILEIT! BASICS	15
Quick Start Guide	15
A Short Guided Tour	16
Options	19
Editing Scripts	20
Compiling	20
Indicators	20
Dialogs	21
Installing into Another Stack	22
Background Operation	22
CHAPTER 3 UNDERSTANDING COMPILEIT!	25
Limitations of CompileIt!	25
Mathematics	25
Accessing Data in HyperCard Fields	26
Data Types	26
Boolean	28
Integer	28
SANE	28
String	29
Record	29
Types of Code Produced by CompileIt!	32
Native Inline Code	32
Library Code	33
Callbacks	34
Purging Your Scripts of Text Callbacks	36
Using ROM Toolbox Routines	40
Procedures and Functions	41

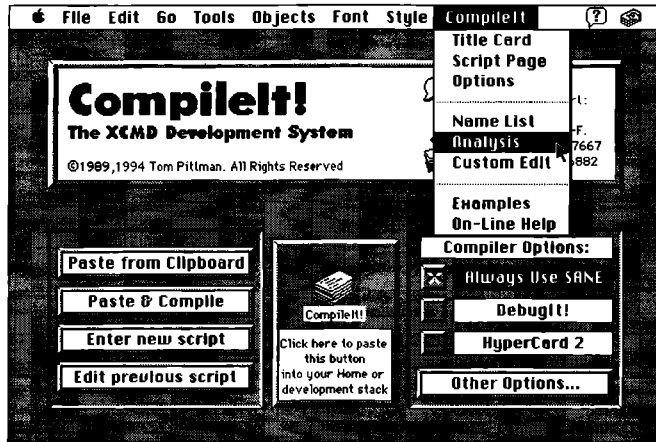
Debugging	42
DebugIt!	42
Invoking DebugIt!	42
The DebugIt! Window	45
The Source Code Listing	45
The Variable Monitor	47
The Controls	49
Some Things You Should Know about DebugIt!	51
Distributing Externals with DebugIt! Attached	53
Analysis	53
CHAPTER 4 THE SYMBOL TABLE	57
CompileIt! Names for Toolbox Access	60
Useful ToolBox Commands and Functions	61
ToolBox Variables and Fields	61
High-Level File Manager Routines	62
HyperCard 2.0 Names	62
SuperCard Commands and Callbacks	62
Color QuickDraw Commands and Functions	63
Custom Symbol Edit	63
K Constant	63
V ROM Global Variable	64
R Record Field	64
X External (XCMD/XFCN)	64
H Text Callback	66
* Binary Callback	66
\$ Raw Inline Code	67
CHAPTER 5 GETTING TO KNOW THE TOOLBOX	69
Bridging The Gap to <i>Inside Macintosh</i>	69
Safe Pointers for Faster Strings	75
Pointers and Handles	79
The Times	82
Handle Chunks	82
Pointers and Handles inside the Macintosh	85
Inside Mac Data Structures	86
More About String Handles	90
Who Owns This Handle	95
Toolbox Calls with Text	99
Shared Variables	100
Array Notation	102

CHAPTER 6 CREATING xWINDOIDS	105
What Else You Should Know about the XCMD Interface	109
 APPENDIX A COMPILEIT! AND OTHER APPLICATIONS	117
SuperCard	118
APPENDIX B SUPERCARD AND COMPILEIT!	119
APPENDIX C COMPILED VS. UNCOMPILED HYPERTALK	123
APPENDIX D ERROR MESSAGES	129
APPENDIX E COMPILEIT! VOCABULARY	143
Supported HyperTalk Vocabulary	143
Unsupported HyperTalk Vocabulary	145
APPENDIX F SUGGESTED READING	147
APPENDIX G MEMORY REQUIREMENTS	149
APPENDIX H UPGRADING TO COMPILEIT! 2.x	151
APPENDIX I WHERE TO FIND MORE INFORMATION ON COMPILEIT!	153
Online Services	153
User Groups	153
CompileIt! Professional Extensions	153
About Updates	154
APPENDIX J TIPS FOR CREATING USEFUL EXTERNALS	155
APPENDIX K INLINE MACHINE-LANGUAGE CODE	159
APPENDIX L INSIDE COMPILEIT!	161
Inside the CompileIt!-Compiled External	161
Compiled HyperTalk Code	162
Inside the Compiler	165
APPENDIX M THE HYPERCARD 2.x XCMD INTERFACE	169
HyperCard Callbacks	171
External Window Events	183
HyperCard Data Structures for Externals	188
 INDEX	191



INTRODUCTION

CompileIt! is the first true compiler for HyperTalk, the programming language for non-programmers on the Apple Macintosh. Before *CompileIt!*, HyperTalk scripts were constrained to run slowly in HyperCard — with no protection from prying eyes! The only alternative was to abandon the ease and comfort of HyperTalk for a conventional language such as C or Pascal. *CompileIt!* makes available the power and speed of these languages, without requiring that you learn a new language and new programming environment.



CompileIt! translates the HyperTalk language into 68000 machine language, and packages the result as a HyperCard external command (XCMD) or function (XFCN) which can then be used from any HyperCard script.

CompileIt! also provides many useful extensions to the HyperTalk language which will give you access to features of the Macintosh far beyond what HyperCard alone provides.

What the Package Contains

Your *CompileIt!* package contains:

- This manual
- The stack *CompileIt!*
- Demo stack
- Batch Compiling Stack
- A folder of optional symbol tables

What Is *CompileIt!*

This section gives you some background as to how *CompileIt!* works. If you know what externals are and what they do, you can skip this section, and go directly to Chapter 2 *CompileIt! Basics*, which will take you through the process of compiling a simple script.

CompileIt! is a tool that allows you to create external commands (XCMDs) and functions (XFCNs) using HyperTalk. Before *CompileIt!*, to create these externals, you had to use a language such as C or Pascal.

Four Reasons for Using *CompileIt!*

- Expand the capabilities of your HyperCard scripts. Use *CompileIt!* and the ROM Toolbox extensions to add new capabilities to HyperCard and similar products.
- Increase the speed of routines written in HyperTalk. Generally, transforming scripts into externals will give you significant speed enhancements.
- Protect sensitive program code from prying eyes. HyperCard is “open,” meaning that others can read your scripts and copy them. Compiled code is not open to this kind of use.
- Learn Macintosh programming. If you are learning how to program the Macintosh, *CompileIt!* can serve as an entry point — allowing you to easily explore the ROM Toolbox.

How to Use This Manual

Users new to *CompileIt!* should read Chapters 1-3 of this manual to gain an understanding of the product and what can reasonably be expected from it. Chapter 5 will introduce you to the Toolbox once you are comfortable with the product.

Experienced *CompileIt!* users should scan Chapters 4, 5 and 6 and the appendices to learn what has changed.

Users new to *CompileIt!* but experienced with Pascal or C will find the information in Chapter 5 invaluable.

Professional programmers will find information of interest to them in the

appendices, in particular, the appendices *INLINE Code* and *Inside CompileIt!* The section *Creating Custom Symbols* in Chapter 4 may also be of interest.

What Is a Compiler?

A compiler is a program that translates another computer program from a high-level, easy-to-read language, such as HyperTalk or Pascal, into the binary numbers that computers understand directly. The “brain” of every Macintosh computer is a powerful processor, the Motorola MC68000 (or one of its big brothers, the MC68020 or MC68030), which understands a complex machine language. Few people understand this machine language, and fewer still are able or willing to write programs in machine language.

Instead, programmers write their programs in high-level, text-based programming languages. The “lowest level” of the programming language is *assembly language*. Assembly language is very close to machine language, but is easier to work with, since it is text, and not number based. Portions of many programs (such as HyperCard) are written in assembly language, since it gives the programmer a great deal of control over what is happening in the machine and allows for greater fine-tuning.

C and Pascal are examples of high-level languages. These languages are more abstract than assembly language, in that you can use any of many commands the language compiler understands, without having to understand the machine or the assembly instructions carried out when that command is executed. Most of these languages are compiled. That is, after the program is written with a text editor, the compiler then scans the program, translating the abstract instructions into specific machine instructions.

Some other programming and macro languages are interpreted. Examples of these languages include the macro languages of such programs as Excel, WingZ, 4th Dimension, and HyperCard. Programs in these languages are actually executed by the “mother” program, and cannot be used outside that program. These languages are very easy to use: they include commands and syntaxes that are natural, and refer to objects (such as records, cells, fields, etc.) that have concrete representations on the screen to the user of the program.

Originally, HyperCard had to *interpret* each statement in a script as it encountered it, figuring out what steps were necessary to execute that line. All this interpreting took time, as HyperCard scanned each line to figure out what had

to be done.

HyperCard 2.0 replaced its interpreter with an incremental compiler which offers better performance under certain circumstances but still requires some “on-the-fly” interpretation.

CompileIt! scans your scripts, translating each command and function into its machine representation, and creates an external command or function of it. This external, being in machine language, is executed much, much faster by HyperCard.

Another benefit of using *CompileIt!* to create externals is that before you begin to compile your external, you can test it just as you test any HyperCard handler.

What Are Externals?

All this is made possible by the decision of HyperCard’s creators to include a feature for extending HyperTalk to include unplanned commands and functions, or to repair HyperTalk problems by replacing existing commands and functions with specially-coded routines in machine language. This capability is the source of unique power in HyperCard, and the ability to use externals has been adopted by developers of many different types of programs, from communications programs to databases and spreadsheets. It allows programmers to extend pre-written software in ways that might not have been foreseen by the software’s original creators.

It was expected that these *external commands* (XCMDs) and *functions* (XFCNs) would be programmed by skilled programmers, using assembly language or C or Pascal. To a large extent this has been the case, but it is unnecessarily restrictive for less skilled users of HyperCard. One of the most significant aspects of HyperTalk is the degree of power it has given to those not skilled or educated in other programming languages. *CompileIt!* completes this empowerment of the individual, by allowing you to use HyperTalk to create your externals.

An external command or function is much like a handler written in HyperTalk or a command or function built into HyperCard. When HyperCard is executing a script, if it encounters a command or function for which there is no handler, it looks in the list of resources that are part of the stack (or of the Home stack) to see if there is an external by that name. If so, the external is executed as if it were a standard HyperTalk handler or a built-in command. (If

an external has the same name as a built-in HyperTalk command or function, then that external takes precedence over the built-in command or function.)

Using externals, you can create the means of making HyperCard do virtually anything that the 68000 processor or Macintosh ROMs can do, within certain limits imposed by the HyperCard environment. Examples include faster and more versatile ways of searching for individual characters or sets of characters in a string of text, creating new types of drawing tools, and bringing to HyperCard many abilities built into the Macintosh ROM.

Your externals, like the handlers you write in HyperTalk, can take parameters (up to the limit of 16 imposed by HyperCard), and though Apple has not released the internal format of HyperCard, your externals can also access data stored in HyperCard fields or in global variables. Messages can also be sent to other handlers in stack and card scripts by a powerful mechanism known as a *callback*. Thus, when writing externals, you have the best of both worlds: the ability to write protected, fast code in machine language, and the ease of use of HyperTalk.

Who Should Use *CompileIt!*

CompileIt! is a development tool for *experienced* HyperTalk scripters. You should be familiar with HyperTalk handlers, messages, and properties before using *CompileIt!* *Suggested Reading* in the Appendices of this manual is a good place to start if you are a beginning scripter.

If you intend to use routines from the Macintosh ROM Toolbox in your compiled scripts, you should be familiar with *Inside Macintosh*. This manual does not provide information on programming the ROM; it only tells you how to use Toolbox routines from your compiled scripts. Again, *Suggested Reading* lists some books in addition to *Inside Macintosh* that can help you.

What *CompileIt!* Does and Does Not Do

CompileIt! allows you to write handlers and functions using HyperTalk, instead of complex languages such as C or Pascal. *CompileIt!* then translates those HyperTalk-based handlers or functions into 68000 instructions for quicker execution by HyperCard.

Many first-time users of *CompileIt!* take some existing scripts, paste them into *CompileIt!* and go. Often the resulting external works fine. In many cases, however, it is not the best way to go.

For example, most handlers in buttons react to the `mouseUp` message that is sent to a button when you release the mouse button over it. You might be tempted to compile such a script for faster processing. However, you should keep in mind that *CompileIt!* generates an external with the name of the handler you compiled — in this case, it creates an XCMD with the name `mouseUp`. Pasting this XCMD into a stack means that any time you release the mouse button on any object (including a card) that does *not* contain a handler for this message the `mouseUp` XCMD will be triggered. This is almost certainly something you do not want to do. Instead, you should break out the tasks performed by the `mouseUp` handler into a separate handler, and then use the `mouseUp` message to call that handler. Compile the separate handler.

Not every script can be made faster simply by compiling it. Scripts that deal primarily with HyperCard structures (buttons, fields, cards, etc.) will not go much faster. Anytime you do something that requires access to the HyperCard file format, you will not realize a speed increase. Some examples of actions that access the HyperCard file format are getting and setting properties, navigating through a stack, sorting cards, and finding.

COMPILEIT! BASICS

This chapter provides a brief overview of *CompileIt!* for both the beginner and the advanced script writer. It is designed to rapidly show you how *CompileIt!* works without going into detail. Going through this hands-on section will help you relate to the concepts covered in the next chapter, *Understanding CompileIt!*

Quick Start Guide

It is important to understand that *CompileIt!* requires some knowledge of the HyperTalk language. If you are new to HyperTalk, you should study one or more of the books listed in the *Suggested Reading* appendix of this manual before proceeding.

The following are the basic steps you will use to compile a script:

1. Make sure you have a backup copy of the *CompileIt!* disk stored in a safe place.
2. Copy *CompileIt!* onto your hard disk.
3. Start *CompileIt!* by double-clicking on its icon from the Finder. (Be sure HyperCard and the Home stack are on your hard disk as well.) The main screen for *CompileIt!* will appear.
4. Click on the **Enter New Script** button. The *Script card* will appear.
5. Type or paste your script into the large text field and click the **Compile It** button.
6. A dialog box will appear listing your HyperCard stacks. Choose the stack where you want your compiled script installed and click **Open**. *CompileIt!* will compile your script; if it takes more than a few seconds, you will hear a beep when *CompileIt!* is finished. As your script is being compiled, you can monitor *CompileIt!*'s progress by watching the gauge at the top of the *Script card*.
7. When *CompileIt!* is finished, you will find yourself in the stack you selected at the start of the compile. If you created an XCMD, its name will appear in the message box. If you created an XFCN, the word `put` followed by the XFCN's name and an open parenthesis will appear in the

message box. Type any required parameters, followed by a close parenthesis (if you compiled an XFCN), and then press the Return key to test your newly compiled script. If your command is specific to a card or background, go to that location first before pressing the Return key.

8. If you get the wrong result, go back to your original source script, correct any errors and compile the script again. Later in this manual, we will discuss *DebugIt!*, which you will find invaluable for locating errors in your scripts.

A Short Guided Tour

CompileIt! is a HyperCard stack. To use it, just open it from within HyperCard, or double-click its icon from the Finder. Always work from a copy of *CompileIt!* — never on the master copy that you purchased. If you haven't done so already, make a copy of *CompileIt!* now and work from the copy. Store the master in a safe place. Similarly, you should first test compiled scripts on a copy of their destination stack.

You can compile any size script from just one handler up to an entire script from a button, field, card, background or stack. However, only the first handler in a compiled script can receive messages from HyperCard or other scripts. Just paste (or type) your script into the *CompileIt!* script card, click on the **Compile It** button, and wait for the script to be compiled — or as happens to most of us, discover you have a programming error, correct it, and try again!

There are extensive help “pop-ups” built into the *CompileIt!* stack (Option-click any button for help). Under System 7.0, Balloon Help is also available. A brief guide to *CompileIt!*'s operation is also included in the stack. There are limits to what *CompileIt!* can compile which are documented in this manual and/or the on-line documentation. This manual also shows how to compile references to the Macintosh Toolbox ROM routines. The examples in the *CompileIt!* stack are also instructive.

You will also note that *CompileIt!* adds a new menu to the standard HyperCard menu bar. This menu lets you quickly move between *CompileIt!*'s various cards.

It is probably easiest to see how *CompileIt!* works by following the development of a simple XFCN from start to finish. We will build an XFCN to calculate the *integer part* of the square root of an integer by subtracting successive

odd integers from it. This is not a fast algorithm, so it is not a very useful XFCN, but its very slowness will help get some perspective on what compilation will do for our scripts.

On the *CompileIt!* distribution disk is a small stack called "Demo." It has just one card and no buttons. The card script has just one handler function `squareroot` as follows:

```
-- calculate ( $\sqrt{y}$ ) by subtracting odd numbers
function squareroot x
  put x into y
  put 0 into ans
  put 1 into oddint
  repeat while y ≥ 0
    subtract oddint from y
    add 1 to ans
    add 2 to oddint
  end repeat
  return ans-1
end squareroot
```

The first thing to do with any script to be compiled is to test it in HyperCard, if that is at all possible. Open the stack, show the message box (Command-M), and type:

```
put squareroot(1000)
```

After a few seconds the message box will be changed to 31, which is the correct answer (as an integer). If it were not, this would be the time and place to debug the script. Despite its limitations, the HyperCard interpreter is much more friendly than any compiler can be. Try a few more numbers, just to convince yourself that it works.

Now open up the card script (by choosing **Card Info...** from the "Objects" menu and then clicking the **Script** button) and copy the script to the clipboard. If there is no "Objects" menu, you may need to type into the message box `set the UserLevel to 5` in order to gain access to HyperCard's scripting level. Next, open *CompileIt!* Click on the **Paste from Clipboard** button. It will take you to the script card of *CompileIt!* If you have any last-minute changes to make to the script before compiling it, now is the time to make them. We will compile it unchanged.

Click on the **Compile It** button. *CompileIt!* will ask you to specify a stack to install the XFCN in; choose the Demo stack. It will take less than a minute to compile this script. You can watch the progress in the meter at the top of the card. You can also see what phase the compiler is in at the top of the card. *CompileIt!* may beep at you when it finishes.

The message box will look like this:

```
put squareroot(
```

Type 1000) and press return.

You will not detect any difference from the first time we tried this (before compiling). This is because there is no difference: the original card script is handling the function call, and the XFCN never sees it. So open up the card script and change the name of the handler, or comment it out, or delete it entirely, then try again from the message box.

You will notice that the new function is so much faster than the original script that you no longer notice any time delay. You actually have to ask for the square root of six digits or more to see any delay at all! If you recompile this script with **Always Use SANE** unchecked, you'll notice that it is even faster. It takes eight digits or more before a delay is noticed. If you are an expert Pascal or assembly programmer, you may be able to squeeze a little more speed out of this XFCN than what you have now, but you would need a stopwatch to notice it.

Notice the second line of the original script:

```
put x into y
```

Although this does not seem to do anything (except perhaps give the compiler a little extra work to do), it has the important effect of separating the character string parameter (x) from the local variable (y) used for arithmetic. Doing so results in a much faster XFCN. Parameters to an external are always passed as character strings. Each time your script refers to them in a math expression, extra code has to be generated by *CompileIt!* to convert them to numbers. Using the above technique results in faster, more efficient, compiled code for parameters that contain numbers.

If you make it a practice to develop your XCMDs in a single stack where they are easily tested (such as the Demo stack we used), then you will need another

tool to copy the finished XCMDs into the stacks where they will be used. A handy stack called "Resource Mover" is included with *CompileIt!* for this purpose.

Options

At *CompileIt!*'s main card, you can set three options that affect the way *CompileIt!* works. Other options are available by using the **Other Options...** button, and these will be discussed later in this manual.

HyperCard 2

This option allows you to use certain symbols (a symbol is any word that has meaning to *CompileIt!*, such as commands, functions, properties, etc.) which, when compiled, result in compiled code that requires HyperCard 2.0 or above. Currently, only **Exit to HyperCard** requires that this option be on. Future versions of *CompileIt!* may enable additional features through this option that will be specific to HyperCard.

This option has no effect on text callbacks (discussed later in this manual) or symbols you specifically install that require HyperCard 2.0. It only applies to built-in symbols that use HyperCard 2.0-specific binary callbacks.

DebugIt!

This option causes *CompileIt!* to attach special code to your compiled external that makes it easier to debug your external. This option is discussed in more detail in Chapter 3 of this manual in the section *Using DebugIt!* Leave this box unchecked for now.

Always Use SANE

This option causes *CompileIt!* to use Apple's SANE (Standard Apple Numeric Environment) routines to perform all calculations, and is discussed later in this manual.

Editing Scripts

CompileIt! works on scripts stored in a field on the **script** card. This means that you can use standard Macintosh editing techniques to put your scripts onto that card. You can paste them from fields or scripts in other stacks, and once a script has been entered into the field, you can use standard Macintosh techniques for editing it: selecting text, then cutting, copying, and pasting it. The four buttons on *CompileIt!*'s first card are provided to make it easier for you to do many of these things.

We recommend that you create a separate stack to contain all your externals. Such a stack gives you a good place to keep both your compiled and uncompiled code, and make notes about any details involved in using your external.

Compiling

While *CompileIt!* is running, you can watch the process to see how much progress it has made on your script. *CompileIt!* makes two passes through the script, and these passes are described below.

You can also change the font and size used in the field that contains your script on *CompileIt!*'s **script** card, by selecting the field, and using the standard commands from HyperCard's menu bar.

Indicators

There are several indicators to help you gauge current progress while *CompileIt!* is compiling. Some of the terms in the discussion below may be new to you; in any case, you don't really need to understand what is going on in each step. This material is provided for informational purposes only.





The purpose of this indicator is to monitor approximate compilation progress, so you can see at a glance how much of the compile is remaining.





This indicator shows the number of bytes of script that have been scanned during Pass 1 and Pass 2. After Pass 2, during the "Add Library" phase, it gives the current number of bytes in the whole XCMD. This number may be more or less than the number of script bytes, depending on the exact script.


CompileIt! puts up a watch cursor with moving hands. The minute hand merely shows that something is happening, but the hour hand shows the compilation phase. This is also reported in the field at the top of the **script** card.


 **Prepare data** The global variables used in compiling are initialized, and the compiler is loaded into memory.


 **Pass 1** The script to be compiled is scanned to determine the types of variables and the names of all the handlers in it. Syntax errors are detected in this pass also.

 **Prepare pass 2** The code generator is initialized, and any string constants are converted to binary code. *CompileIt!* watches for duplications so that space will not be wasted on them.

 **Pass 2: Gen code** The script to be compiled is scanned again, and binary code is generated for all the commands and expressions. This is usually the longest part of the compile.

 **Add Library** *CompileIt!* has a library of functions to aid in evaluating chunk expressions and other operations that do not translate easily to machine language. Only the library routines that are actually used are added to an XCMD being compiled.

 **Link functions** Handlers and library routines are linked to the commands where they are called.

 **Build resource** Finally, the binary code is converted to a resource and added to the target stack.

Dialogs

There are two dialogs which require user intervention while running *CompileIt!*

One dialog is normally seen each time a compile is started. It asks you to select the stack where the XCMD is to be installed, using the standard file dialog box.

When *CompileIt!* aborts compilation on a programming error, it leaves a message in the script field (see Appendix D, Error Messages), then puts up a dialog to report the error. In the dialog, the next symbol to be scanned from the script is mentioned in quotes.

Installing into Another Stack

When *CompileIt!* starts to compile your external, you will be asked to locate the stack into which you want the external to be installed. You can install your new external into any stack except for Home, *CompileIt!* itself, any stack that is “in-use,” or any other currently open stack.

If you do not wish to be asked each time for the name of a stack, you can go to the **Other Options** card (by clicking the button of that name on the first card, or selecting Options from the *CompileIt!* menu), and set the name of the stack into which you want all your externals to be installed. In the lower left corner of that card, you will see a field labelled **Always Install in Stack...** which is empty. Clicking on this field summons a standard **Open File** dialog box which lets you locate a stack. The name of that stack will be recorded in the field, and all new externals will be installed in that stack with no intervention from you.

We recommend that you maintain a library stack of all the new externals you create, as a handy place to store your original code, along with notes about using the external. However, this should *not* be the stack into which you always install your new external. Externals with errors in them — especially those that use the ROM routines — can wreak havoc on your stacks, perhaps destroying them. Instead, create a stack you use for testing purposes, and install all new externals into that stack. Use Resource Mover to transfer completed externals to your library stack.

Background Operation

CompileIt! can compile in the background under all versions of HyperCard. If you are compiling a very large script under HyperCard 2.0 or later, you may find it convenient to “switch-out” to another application to work on something else. *CompileIt!* will notify you when it finishes by flashing a small stack icon over the Apple menu, unless you have set up a batch compile stack to compile multiple scripts.

Even if you are not running under Multifinder or System 7.0, or are using an earlier version of HyperCard, you can use desk accessories while you compile.

CompileIt! is intelligent about how it works in the background. It will notice if nothing is happening in the foreground and run at full speed during those times (for instance, when you are reading a file in a word processor or away from your desk for a bit).

UNDERSTANDING *COMPILEIT!*

This first half of this chapter discusses data types, limitations, the types of code that *CompileIt!* produces, and text callbacks. The second half introduces *DebugIt!* and the Analysis card, which are used to find programming errors.

Limitations of *CompileIt!*

Mathematics

SANE (Standard Apple Numerics Environment) is a set of routines built into every Macintosh. SANE ensures that calculations involving “real” numbers (very large or small numbers, or numbers with fractional parts) always give the same level of accuracy regardless of what application or machine the calculation is performed in/on (rounding errors, while impossible to eliminate entirely, are ensured to be consistent). However, SANE is much slower than integer arithmetic (calculations involving NO decimal portion) and is unnecessary for such calculations (accuracy is a constant). HyperCard almost always uses SANE even when it is unnecessary, while *CompileIt!* gives you a little control in the decision.

On *CompileIt!*'s first card is a button labeled **Always Use SANE**. If you have checked this button, then *CompileIt!* will use Apple's SANE for *almost* all math operations. The exceptions are cases where *CompileIt!* knows that integers are used: such things as indices in Repeat loops and ROM calls that require integer input. SANE is slower than integer mathematics, so you might want to leave this button unchecked. Don't worry, though: if *CompileIt!* sees that you are working with a floating point number it will use SANE automatically regardless of the setting of this button. The purpose of this button is to safely enable you to get the fastest possible code — if this option is off, *CompileIt!* will use faster integer math when in doubt; if on, slower SANE math will be used when *CompileIt!* is in doubt.

If you are working with SANE unchecked there are a few situations where *CompileIt!* may choose the wrong data type. For example, consider the following line of HyperTalk:

```
add (item 2 of x) to item 4 of y
```

CompileIt! has no idea what type of data is in the items at compile time. If the

SANE checkbox is *unchecked*, the integer data type will be chosen. If your script contains HyperTalk similar to the above example and you still don't want to compile with **Always Use SANE** checked, you can ensure that the SANE data type is chosen by using the exponential (^) operator as in:

```
add (item 2 of x)^1 to item 4 of y
```

Using ^1 does not result in additional code, it is simply a hint to *CompileIt!* to use the SANE data type.

In general, you won't need to worry about the details of using SANE: when *CompileIt!* encounters an operation that involves a floating point number (i.e., x/y or $x+3.345$), the SANE routines are used automatically even if the **Always Use SANE** box is unchecked.

Accessing Data in HyperCard Fields

Your compiled scripts can refer to data in parameters, local variables, global variables, and data in fields on the *current card* only. From within your external, HyperCard will not let you refer to data in fields on cards other than the current card. This means that you cannot use a command like: get line 1 of field 5 of card 10. A work-around is to use the go command to go to other cards first as in:

```
go to card 10
get line 1 of field 5
```

Another work-around lets HyperCard do the work for you using the do command and the value function, as in:

```
do "get line 1 of field 5 of card 10"
get the value of "it"
```

You may also want to read *Compiled vs. Uncompiled HyperTalk* in the Appendices, which discusses some of the subtleties of compiled HyperTalk.

Data Types

One of the things that makes programming in HyperTalk easy and Pascal hard is that HyperTalk hides all the different machine representations of data from the programmer. Everything in HyperTalk is a string of characters. The

term “hides” is used because the arithmetic is not done on strings of characters inside the machine. You just aren’t aware of all the conversions going on all the time. The conversions are necessary, but you don’t have to think about them; they are automatic.

Once in a while HyperCard shows its hand a little bit. As an example, type into the message box:

```
put 20000000 / 100
put 20000000 div 100
put 2000000000000000 / 100000
put 2000000000000000 div 100000
```

Since the division is exact (no remainder for div to discard), all four lines should get the same result, but HyperCard 2.0 tells you that the result of the last statement is -2147483648 and versions of HyperCard less than 2.0 tell you that “2000000000000000 is not the right type for div.” What has happened is that HyperCard is doing its arithmetic using integers as much as possible, since these are the fastest arithmetic in the 68000, using machine language hardware. When HyperCard notices that the numbers exceed its integer size, or if fractions are possibly involved (as in the case of “/” division), it switches over to SANE floating point. SANE is executed in hardware on the Mac II, and other processors with a 68881 or 68882 floating point co-processor, but in a standard Macintosh it is emulated in software, which is rather slow.

An integer like 20000000 fits into 32 bits easily, but 2000000000000000 does not.

Why this emphasis on numeric data types? Just this: while HyperCard may have a couple of surprises for the unwary programmer, most of them can be caught at run-time by examining the data and choosing the most appropriate data type. Compiled HyperTalk has a lot more surprises.

CompileIt! attempts to choose an appropriate data type for your variables when it compiles the script, without looking at the data (since, of course, no data is available yet), by seeing to what use the variables are put. It is almost impossible to tell ahead of time whether decimals or fractions are involved, so the safest code would use SANE all the time; this is generally what happens if you have the **Always Use SANE** button on **CompileIt!**’s first card checked. This is slow for most applications where integer arithmetic is perfectly adequate; so if your external is using integer math only, leave this button unchecked.

Another data type problem comes from odd (but perfectly legal) expressions like:

```
char 3 to 5 of (1234*9876) + 7
```

(which has the value 193). The sub-expression inside the parentheses is an integer multiplication. To this is applied a chunk expression evaluation to extract the third, fourth, and fifth digits *as a character string*, which once again is treated as an integer and added to 7. *CompileIt!* will give the correct result for this expression, but in the process it will be converting the integer product to a character string to extract the three characters from it, then converting that sub-string back to integer to add 7. All this conversion takes time — indeed, it takes about as long as HyperCard would have taken, because HyperCard must do the same kinds of conversions itself.

There are basically five data types that make sense in compiled HyperTalk. These are all modelled in HyperCard as character strings, but for the best performance, and for correct interface to the ROM Toolbox routines, *CompileIt!* uses the conventional 68000 representations for the five data types. The five types are *Boolean*, *integer*, *SANE*, *string*, and *record*. Additional data types are available when working with the Toolbox. The additional data types are presented later in this manual.

Boolean

Boolean data has only two abstract values, *true* and *false*. These are modeled in HyperCard as the character strings “true” and “false”, but in normal 68000 code they are the two values of a single bit (1 or 0). *CompileIt!* uses the single bit representation, and converts to and from character strings as needed to communicate with HyperCard.

Integer

All integer math is 32-bit ($\pm 2^{31}$).

SANE

All SANE math is 80-bit ($\pm 1.1 \times 10^{49}$).

String

Strings in HyperCard can be any number of characters, up to the total of available memory. Fields are limited to about 30,000 characters, but variables have no such limitation. *CompileIt!* supports this data format fully.

In order to make the memory space used by string data available when the strings are no longer in use, XCMDs compiled by *CompileIt!* include a procedure called a garbage collector. This takes a little longer than a program where the programmer knows exactly when a string is no longer in use and releases its storage immediately. Pascal programmers are required to do this explicitly in their XCMD. If they forget, memory slowly fills up with strings that are no longer in use, and they have to quit HyperCard to get rid of them. With *CompileIt!* you don't have to worry about forgetting — or worse, disposing of a string before you are finished with it (a sure bomb). However, if your XCMD terminates abnormally with an error in a callback to HyperCard, there may be undisposed of strings left. The best thing to do in such a case is to quit HyperCard as soon as practical, and restart.

The ROM Toolbox routines require strings in a different format than does HyperCard. Sometimes called “P-strings” for historical reasons (they achieved popularity with the UCSD P-System that was the predecessor of Apple Pascal), these strings are limited to a maximum of 255 characters. With such a limited size, they do not require garbage collection. *CompileIt!* is aware of the differences between these two string formats, and makes appropriate conversions. However, whereas P-strings can have any characters in any position, HyperCard strings cannot contain any null characters (a null marks the end of the string). Although not normally a problem, Desk Accessory names returned by various Toolbox routines all begin with a null, so the conversion makes their names appear to be empty. The only way around this problem involves the explicit use of P-string data types in shared variables or on the heap (see Chapter 5 *Getting to know the Toolbox*), and writing a short handler to replace the null with something else.

Record

Records in HyperCard are represented as variable-length lists of words, items, and lines, packed into single strings. This is fairly easy for non-programmers to understand, but it is difficult for programming languages to extract or change components. Conventional programming languages use fixed-length integers and P-strings, packed into a block of memory also of a fixed length.

The machine language to access such data structures is very efficient.

Most of the data formats in *Inside Macintosh* are defined as records in the Pascal language. It is not really necessary to understand all of the Pascal language to read these records; it is sufficient to recognize the keywords **record** and **end** which act like brackets around all of the data contained in the record (including possibly other, nested records). Any item inside the record is accessed by attaching its name with a dot (".") to the record reference. Thus, for example, with the record definition (from *Inside Macintosh*, I-202):

```
BitMap = record
  baseAddr: Ptr;
  rowBytes: integer;
  bounds: Rect
end
```

If you have a pointer to a BitMap in the variable B, you can access the integer value rowBytes by the Pascal equivalent of a chunk expression, B@.rowBytes. This is read approximately as, "the rowBytes item of the BitMap record pointed to by B."

A good Pascal compiler rigidly enforces correct data type management by the programmer. This is mostly a protection against programming errors, and results in better productivity than un-typed and weakly typed languages like C and assembly. Because HyperTalk has no types at all in the language, *CompileIt!* can only make a best guess at appropriate data types, and automatically convert between the types as necessary.

In the case of Toolbox routines and HyperCard callbacks, the data types are well-defined (as indeed they must be for Pascal programmers). Within the compiled XCMD, types are assigned to local variables according to their usage. If they are used in string operations, they will always default to a string type as the safest. Otherwise, variables passed to integer Toolbox routine parameters or used in integer arithmetic will likely be assigned the type integer. Variables used in Boolean expressions will likely be assigned the type Boolean, etc.

The special HyperTalk variable `it` gets reused for whatever is handy so often that *CompileIt!* will reassign its type every time you put a new value into it or get a new value. This can lead to inconsistencies in the case of repeats or if-then-else commands. The following script has two such inconsistencies:

```
get true -- set the type of 'it' to boolean
repeat until it is empty
  if whatever then get empty -- set type of 'it' to string
  else get 5 -- set the type of 'it' to integer
end repeat -- the type of 'it' is now inconsistent
```

The “then” part of the if-then-else gets an empty string into it, setting its type to string, but the “else” part gets the integer 5 (setting its type to integer). If any use of `it` is to be made after the two branches rejoin, there is no way for the compiled code to know which type is appropriate. This could be resolved by changing the `get 5` to `get "5"`, so that the one-digit character string `"5"` (type string) is stored in the variable `it` instead of the integer 5. That will not invalidate any intermediate use of `it` such as for arithmetic, since *CompileIt!* will make the necessary conversions automatically now that the type is consistent through the if-then-else part of the loop.

However, there remains another inconsistency around the loop. The line before the repeat command gets a Boolean value, but the if-then-else leaves a string type in `it`, and the condition on the repeat is explicitly testing for a string condition, namely, the empty string. The Boolean value is not a string, but *CompileIt!* will convert it, probably yielding either the string `"true"` or the string `"false"`, depending on what HyperCard makes of the empty string interpreted as a Boolean value. Neither of these two strings is empty, so the loop will never terminate. Again, enclosing the word `"true"` in quotes will yield a consistent string type all around the loop, and the test for empty will then be valid and functional.

Other than watching for and avoiding this particular problem, you do not need to be overly concerned about data types. A good rule of thumb is to use local variables for all arithmetic and loop index (with-variable) operations, and to avoid concatenation and chunk expressions on the same variables that are used for arithmetic.

To avoid misunderstandings, you can “declare” variables to be of a certain type. For example, if you are going to be using one particular variable to hold integers only, you could put a line such as this:

```
add 0 to myVariable
```

somewhere in your script. That way, *CompileIt!* will know what kind of variable it is. If a variable is used as a parameter to your external, you can make sure *CompileIt!* knows what it is by first copying it into a local variable,

then adding 0 to it, as with these two lines:

```
put parameter1 into myVariable  
put myVariable + 0 into myVariable
```

Similarly, you can tell *CompileIt!* that a variable is to contain a floating point number by using the line:

```
put myVariable^1 into myVariable
```

In neither case do these statements add size or processing time to your external.

If you have a number that is a floating point number, and wish to convert it to an integer for faster calculation (losing, of course, the fractional part of the number in the process), you can use the `Round` or `Trunc` functions. Further operations on variables containing the results of these functions will use 32-bit integer math.

Types of Code Produced by CompileIt!

CompileIt! produces four different types of code, depending on the type of source it is dealing with. Each of these four types of code yields a different level of performance (i.e., speed) in your external. The four types of code are: *native inline code*, *library code*, *binary encoded callbacks*, and *text callbacks*. As a user of *CompileIt!*, you don't need to worry about these different kinds of code, except that they do produce dramatically different levels of performance in your externals.

Native Inline Code

Native Inline Code is the fastest. On a 68000-based Mac SE or Classic, this kind of code runs at about 400,000 instructions per second, and it takes some 2-8 instructions to do a command or calculate a simple expression value in native inline code. If your entire script compiled to native code, it would run 5,000 to 10,000 times faster than interpreted HyperCard. The following types of operations generate native inline code:

- Integer addition, subtraction, multiplication, and compare
- Integer division or multiplication by a constant power of 2 (e.g., 32 or 512)

- Single character compare (excluding chunk expressions)
- The And, Or, and Not operators
- All forms of repeat, including Next Repeat, and Exit Repeat
- All forms of If, with or without Else
- Calling functions compiled in the same script
- Recursion (i.e., calling a function from within itself)
- Passing messages to local handlers (i.e., those compiled within the external)
- Exit or Return from a handler or function that uses no string variables
- Put number or integer expression into local or shared integer variable, or conditional expression into Boolean variable
- Get integer or Boolean expression
- Array, pointer, and record notation (e.g., `handle@@.integerType[x]`)
- NumToChar, CharToNum, Abs, Max or Min (with integer params), ParamCount function
- The ROM Toolbox functions: BitAnd, BitOr, BitXor, LoWord, HiWord

Library Code

Library Code also executes very fast. Because compiled Library code executes hundreds of instructions for each command, it is not as fast as native inline code. This is used for commands and expressions that do not convert directly to native machine code, but are relatively simple to do with repeat loops in specially coded routines, such as 32-bit integer division and string operations. The repetitive part of the code will typically take about as long for the compiled XCMD as for the HyperCard interpreter, but the interface is in native inline code, eliminating the run-time analysis of what to calculate. Short strings and arithmetic will tend to run somewhat faster than HyperCard, but long strings may take about the same amount of time. Some commands and expression operators depend on ROM Toolbox calls for part of the processing; these are included here, since the timing for them is comparable to compiled library code. The following kinds of operations produce Library Code:

- String/ number conversion
- SANE arithmetic
- String compare
- Integer division other than by a power of 2
- String concatenation
- Delete
- Chunk expressions
- Exit and Return from handler or function
- ROM Toolbox calls
- Put a string value into a local variable
- Put a string value into a chunk
- Get a string value
- Put, Get
- Length, Number (of chunks), Result, Param, Params, Average
- One-parameter HyperCard math functions: atan, average, cos, exp1, exp2, ln, ln1, log2, min, max, random, round, sin, sqrt, tan, and trunc

Callbacks

As we have explained, *CompileIt!* works by translating HyperTalk commands into their machine language equivalents, so that HyperCard can perform these tasks much quicker. *CompileIt!* includes in code or its symbol table machine-language counterparts for a large percentage of HyperTalk's commands. However, *CompileIt!* does *not* include machine language representations of *all* of HyperCard's commands. Many of these are handled by a mechanism built into HyperCard's external facility called the "text callback."

Callbacks were built into HyperCard to allow external routines to have HyperCard do some of their work for them. Examples include getting text from fields, executing commands such as `Find`, converting data into a form that HyperCard can understand (i.e., text). Callbacks are very useful: since Apple has not made the data structures that HyperCard maintains in RAM and on disk public, there is no other mechanism for externals to get at HyperCard data.

However, callbacks take time. A text callback is generally sent to the current card, from which it must make its way up the complete hierarchy of HyperCard message passing, then be executed in the same manner as a HyperCard command or function. Furthermore, text callbacks must be translated into a text message by the external before they can be sent to HyperCard, which takes more time. Ideally, you will want to write your *CompileIt!* externals so that they include virtually no text callbacks. This is not always possible. This section will help you understand what the various callbacks are and how to identify them and remove them from your scripts so that your external is as fast as possible.

There are two kinds of callbacks supported by HyperCard and by *CompileIt!* The first type, Binary encoded callbacks, are very fast, because the message does not need to be converted to text. These callbacks include:

- Number and string conversion
- String search and compare operators (used for Offset or Contains)
- Global variable access (that is, changing a global variable with the Put command, or referring to it in an expression)
- Field access on the current card (that is, Getting or Putting text into a field)
- The script abort operator (in HyperCard 2.0, used for Exit to HyperCard)
- Script access utilities in HyperCard 2.0.

The other kind of callback is the Text message. These callbacks are at least as slow as the original command in HyperTalk, and because of the overhead of having the external send the message to HyperCard, can actually be slower.

The following result in text callbacks:

- The Within operator
- Accessing properties
- Put, into the message box, or Put with no destination (message box assumed)
- Reference to the It variable, after using the Convert, Ask, Answer, or Read commands

- References to handlers, functions, and other externals not compiled with your script (unless defined as an Inline call to another external with Custom Symbol Edit)
- Commands, functions, and properties that *CompileIt!* does not know about

Purging Your Scripts of Text Callbacks

This section will show you how to identify callbacks in your scripts and provide suggestions on how they might be eliminated. As discussed in the Debugging section of this chapter, you can use the **Analysis** button on *CompileIt!*'s **Script** card to help you see, once your script has been compiled, what callbacks are used.

Here is an example of a very slow script, and some ideas for how to make it faster:

```
on demo0
  global x
  repeat while the mousetloc is within the rect of card button 3
    add item 6 of card field 3 to x
  end repeat
end demo0
```

There are three text callbacks on the repeat line of this script. That means that every time around the loop three messages are sent back to HyperCard to interpret. At least one of these is never changing, so it can be moved out of the loop for a significant improvement:

```
on demo1
  global x
  put the rect of card button 3 into r3
  repeat while the mousetloc is within r3
    add item 6 of card field 3 to x
  end repeat
end demo1
```

Although convenient, the `within` operator is not very fast in compiled code because it is sent back to HyperCard to be interpreted. You can write a short function that will run much faster, if you convert the coordinates to integer first:

```

on demo2
  global x
  put the rect of card button 3 into r3
  put item 1 of r3 + 0 into r3left
  put item 2 of r3 + 0 into r3top
  put item 3 of r3 + 0 into r3right
  put item 4 of r3 + 0 into r3bottom
  repeat while iswithin(the mouseloc, r3left,r3top, r3right,
r3bottom)
    add item 6 of card field 3 to x
  end repeat
end demo2

```

```

function iswithin mloc, lf, tp, rt, bm
  get first item of mloc + 0
  if it<lf or it>rt then return false
  get second item of mloc + 0
  if it<tp or it>bm then return false else return true
end iswithin

```

Notice that adding zero to a value tells the compiler that the value is numeric; it will then convert it to integer immediately and use the integer value for later computations. Although this script is much longer than the original script, it is also considerably faster, since we have eliminated all but one of the callbacks within the repeat loop.

The function `iswithin` uses no string variables — the only strings are the parameter `mloc` and temporary values extracted from it — so its call and return are also very fast. The comparisons are all integer compares, and the function returns a Boolean value that can be tested very quickly by the inline code of the repeat in `demo2`.

Within the repeat there is a chunk expression referring to a field. The field value does not change during the execution of the repeat, so we can safely move this part of the computation outside the repeat and pre-convert it to integer (there are no changes to the function `iswithin`, so it is omitted here for brevity):

```

on demo3
  global x
  put the rect of card button 3 into r3
  put item 1 of r3 + 0 into r3left

```

```

put item 2 of r3 + 0 into r3top
put item 3 of r3 + 0 into r3right
put item 4 of r3 + 0 into r3bottom
put item 6 of card field 3 + 0 into i6
repeat while iswithin(the mousetloc, r3left, r3top, r3right, r3bottom)
    add i6 to x
end repeat
end demo3

```

The global variable `x` is being used twice every time around the repeat, once to get its current value, and a second time to put the new value back. Global variables, like fields, are passed back from HyperCard as text strings, so there is also a string-to-number and a number-to-string conversion going on each time around the repeat. We can save considerable time by moving this conversion outside the repeat loop, provided that there are no callbacks left in the loop that depend on the global variable `x` (and there are none in this example):

```

on demo4
    global x
    put x into localx
    put the rect of card button 3 into r3
    put item 1 of r3 + 0 into r3left
    put item 2 of r3 + 0 into r3top
    put item 3 of r3 + 0 into r3right
    put item 4 of r3 + 0 into r3bottom
    put item 6 of card field 3 + 0 into i6
    repeat while iswithin(the mousetloc, r3left, r3top, r3right, r3bottom)
        add i6 to localx
    end repeat
    put localx into x
end demo4

```

This time we did not need to add zero to the value of `x` to ensure that it came out numeric, since the compiler can see that its main use is to have `i6` added to it. If you are not sure, take a look at the **Analysis** card to see what type *CompileIt!* assigned to the variable. In any case, adding 0 to a variable has no cost in processing time.

We have not yet gotten rid of the last text callback in the repeat loop. It happens that there is a ROM Toolbox call that returns the coordinates of the mouse as a `Point`. A point is a 32-bit integer, where the high-order 16 bits are the vertical and the low-order 16 bits are the horizontal coordinate. We can

use the fast Toolbox calls `LoWord` and `HiWord` (which are compiled inline instead of jumping off to the ROM) to separate these:

```
on demo5
  global x
  put x into localx
  put the rect of card button 3 into r3
  put item 1 of r3 + 0 into r3left
  put item 2 of r3 + 0 into r3top
  put item 3 of r3 + 0 into r3right
  put item 4 of r3 + 0 into r3bottom
  put item 6 of card field 3 + 0 into i6
  repeat while mouseiswithin(r3left, r3top, r3right, r3bottom)
    add i6 to localx
  end repeat
  put localx into x
end demo5

function mouseiswithin lf, tp, rt, bm
  GetMouse mouseVH
  put LoWord(mouseVH) into hh
  put HiWord(mouseVH) into vv
  return hh>lf and hh<rt and vv>tp and vv<bm
end mouseiswithin
```

The performance is tricky to judge in a loop like this that depends on user interaction. The results below were arrived at by dividing the number of ticks the XCMD takes to return by the difference in `x` (assuming the value in item 6 is 1) to get the average number of ticks around the loop once:

```
-- this is the benchmarking script.
on mouseUp
  global x
  put the ticks into oldt
  put 0 into x
  put 1 into item 6 of card field 3
  repeat until x ≠ 0
    beep
    demo0
  end repeat
  put "Average time is" && (the ticks - oldt) / x
end mouseUp
```


Although there is a lot of variability, depending on disk latency, size of memory, and other factors too complex to mention here, the differences in performance time on a Mac Plus were significant. The same test on an SE/30 got corresponding improvements.

	Plus Time	SE/30 Time
demo0 Uncompiled	6.0	1.8
demo0 compiled	10.0	2.4
demo5 compiled	0.12	0.015

Using ROM Toolbox Routines

If *CompileIt!* could only be used to speed execution of standard HyperTalk commands, it would be useful, but would not really give you a lot more than HyperTalk itself gives you. One of the things programmers do with externals is provide access to capabilities not normally present in HyperCard. This is done, as is most Macintosh programming, by using the set of routines built into the Macintosh ROM, and contained in its System File. This set of routines is commonly called the Toolbox, and contains support for all the things that make the Macintosh special: fast QuickDraw drawing, control of menus and dialog boxes, resources, and the like. *CompileIt!* allows the HyperTalk programmer to explore the Macintosh Toolbox, and make use of it when creating externals.

This section will help you create externals that use the Toolbox. It is not, however, a complete discussion of the Toolbox — Apple has taken 6 volumes of over 3000 pages to do that! Instead, we will assume that you have, or at least plan to get, *Inside Macintosh*, and one or more of the books on programming the Macintosh listed in the *Suggested Reading* appendix. In particular, it is recommended that you have at least volumes 1, 2, and 4 of Apple's *Inside Macintosh*. Volume 3 is little more than an index (since replaced by the *Inside Macintosh XRef*), but volume 5 is required if you are programming for a color Macintosh, and volume 6 is essential for System 7. Programming the Toolbox is difficult enough with these books; it is virtually impossible without them.

Procedures and Functions

When you look through *Inside Macintosh*, you will discover that the two types of routines described are always identified as either Procedures or Functions.

In general, *Inside Macintosh* procedures can be thought of as being the same as HyperTalk command handlers: they perform some action, and optionally can take several arguments. Most of the time, you will use these procedures in *CompileIt!* just as you do normal HyperTalk commands.

You should note, though, that Procedures are shown in their listings in *Inside Macintosh* with parentheses around their arguments. You should not include these parentheses when calling a Toolbox Procedure.

Toolbox functions behave the same way as do HyperTalk functions: they take zero or more arguments, and return a value to the line that called them.

This is not always the case, however: for example, the `getFNum` procedure on page I-223 of *IM* returns the number of a named font. This procedure takes two arguments: the first is the name of the font, the second is the name of a variable, defined as `Num` in *IM*. The `getFNum` procedure works as a function does: it puts into the variable `num` the number of the font whose name was in the first variable. You can identify these types of variables in *IM* by the name `VAR` in front of the parameters.

To understand and use *Inside Macintosh* routines, you should be comfortable with Macintosh memory management, and the concept of pointers and handles. In short, a pointer is a location in memory — typically represented as a number stored in a variable — that in turn points to another address in memory, which actually contains the data you are using. A handle is one step removed from a pointer: instead of pointing at actual data at a location in memory, it points instead at a pointer. Many of the Toolbox routines require either pointers or handles to refer to the objects they work on. For more information on pointers and handles, you should consult Appendix G of this manual, which discusses using them in relation to HyperCard strings. Good discussions of Macintosh memory management can also be found in the Chernicoff and Knaster books mentioned in Appendix F.

You should also be somewhat familiar with *Inside Macintosh* data structures. As discussed earlier, you don't necessarily need to know Pascal in order to understand how to get at the various *IM* records. *CompileIt!* is aware of most

IM data structures, and if you follow the example given earlier in this section, you should be able to get at most data stored in these records.

Debugging

The easiest way to debug a compiled script is with DebugIt! (discussed below). In addition to DebugIt!, *CompileIt!* offers a special flag that you may find useful.

The \$ symbol is just like a comment (--) if the DebugIt! option is turned off. Lines in your script preceded by this symbol will be compiled only if the DebugIt! option was on when you compiled your script. You must follow this symbol with a valid command like "put" or you will get a compiler error message. You might use this option to add extra code for looking at variables or controlling loops during testing, or for locking handles whose contents might be upset by DebugIt.

DebugIt!

DebugIt! is a source level debugger for *CompileIt!*-created externals (XCMDs/XFCNs) that greatly facilitates the location and correction of programming errors. One of the most difficult and time-consuming areas of programming is locating programming errors.

With DebugIt!, you can view your original source code at runtime along with all variables. DebugIt! lets you execute the external a line at a time or continuously until some predetermined point. Variables can be viewed and edited "on-the-fly" and the intrepid can even view and edit RAM if they wish. Error detection and correction are greatly facilitated because you can actually see what is going on when the errors occur rather than having to resort to guesses and working backwards from a wrong result.

Invoking DebugIt!

Rather than forcing you to test your externals under a protective shell as some debuggers do, DebugIt! is designed to work "in context" with your external. If the **DebugIt!** checkbox is checked (either on the **Title** card or on the **Script** card of *CompileIt!*) then *CompileIt!* will build DebugIt! into your external. While this adds 39K or more to the size of your external, it means DebugIt! will be available anywhere you test your external, even in environments other than HyperCard.

Because DebugIt! is not tied to HyperCard, throughout the remainder of this section specific references to any application have been avoided except where required. Instead, the application where you will be testing your external is referred to as the "host application." This means any application that supports the HyperCard 1.0 XCMD interface, or better. Some host applications do not support the entire 1.0 XCMD interface (certain host applications do not support the callbacks `evalExpr`, `sendHCmessage`, and `sendCardMessage`). DebugIt! should still work with these applications but some functions may not be available or may not work as expected (the **Refresh** and **SendMsg** buttons in particular are dependent on these callbacks). If an application has limited support for the 1.0 XCMD interface and conforms with the way Claris says these things should be done, then the features should just fail to work rather than cause a system crash.

Obviously, an external with DebugIt! attached will require more memory than without DebugIt! You should allow at least 100K of memory for each currently executing external that has DebugIt! attached. If **CompileIt!** reports (via the **Options** card or upon entering the **Script** card) that it can compile in its "Fast" mode with DebugIt! checked, then you can reasonably assume you will have enough memory available to test the same script with DebugIt! after it is compiled.

To use DebugIt!, just check the **DebugIt!** checkbox and compile your script as you normally would. When you execute your new external, the DebugIt! window will open and execution will halt before the first line is executed (assuming you are testing in HyperCard or SuperCard during the same session as you compiled the script). At this point, you can set any breakpoints, single step through the external, etc. We'll explain your options a little later.

Since you may have multiple externals being tested at the same time and each may have DebugIt! attached, you need a way to control which ones will open their debugging windows. This is done through a global variable called "HyperDebugIt" (note: no exclamation point). **CompileIt!** will initialize this global for you during your compile but you'll need to initialize it yourself if you quit and come back to test later — unless you have set breakpoints in the external, and are content to open the window at the first breakpoint.

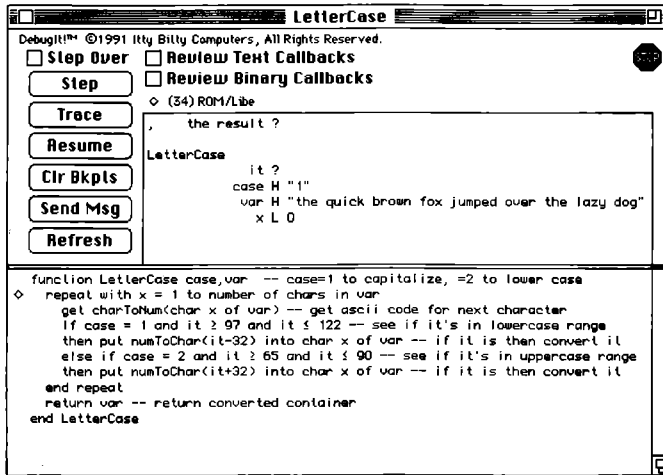
The following logic is used to evaluate the contents of the HyperDebugIt global and control the debugging windows. If the first word of HyperDebugIt is TRUE, then the rest of the global is ignored and any external with DebugIt! attached will open its window when executed. If the first word of HyperDebugIt is FALSE, then no externals will open their debugging windows unless they hit a breakpoint. If the first word is not TRUE or FALSE then the items and words in the global are examined. If the names of any externals that have DebugIt! attached are found in HyperDebugIt then their debugging windows will open when they are executed. *CompileIt!* puts the name of the external into HyperDebugIt after compiling an external with the DebugIt! option checked. If nothing is in HyperDebugIt then obviously no external has its name in it, so no debugging windows will open, which has the same effect as FALSE.

Regardless of the values in HyperDebugIt, the debugging window will always open if a breakpoint is encountered during execution. Breakpoints will be discussed in a moment.

Under HyperCard 2.x, DebugIt! will place additional information at the end of the HyperDebugIt global. A space should always precede this extra information and nothing should follow it. This information is used internally by DebugIt! and should not be removed or edited. This extra information is used by DebugIt! to conform to rules that govern external windows under HyperCard 2.x while still maintaining compatibility with other, non-HyperCard, host applications.

The DebugIt! Window

Below is an illustration which shows the DebugIt! window. It may be helpful to go to the DebugIt! help card in *CompileIt!* (the card after the on-line help card), which contains an annotated (and slightly animated) copy of this same screen shot.



The window consists of three main parts; 1) the source code listing for the external, 2) the variable monitor, and 3) the controls. Let's look at each separately.

The Source Code Listing

This is the original source code that you compiled with *CompileIt!* You cannot edit the source at this point since it is only there for reference (the machine code that *CompileIt!* created from your source code is what is really being executed). You can do three different things with this source.

- 1) You can click on any line within a handler (the line will highlight) and an analysis of that line will appear above the variable monitor. The analysis will always begin with a number in parentheses like "(16)". This value is the size of the actual machine code that line represents (in bytes). If nothing else is listed then the line is made up entirely of 68000 machine code. Other items that may appear as part of the analysis are:

BinCbk - This line has one or more binary callbacks

TxtCbk - This line has one or more text callbacks

ROM/Libe - This line has one or more ROM or library routine calls

NumCvt - This line has one or more number / string conversions

Glo/Fld - This line has one or more references to a HyperCard field or global variable

Bkpt - There is a breakpoint set for this line

Text callbacks are also listed as part of the analysis on the **Analysis** card in *CompileIt!* and are the slowest kind of code *CompileIt!* produces (sometimes slower than before they were compiled). A small "T" will appear next to any lines that contain text callbacks as well.

Binary callbacks are not as bad as text callbacks (global variable and field references are examples of binary callbacks). Number conversions take about as long as binary callbacks and ROM/Library routines; both are a little bit slower than native 68000 code.

You can use the analysis to pinpoint areas of your scripts that could be optimized if speed is an issue for you. This information is also useful if you are trying to write code that has no callbacks at all (callbacks are not allowed in filter functions and VBL tasks for example).

- 2) You can set breakpoints by clicking in the margin before any executable line (executable lines are any lines that appear inside of a handler including the "end" handlername statement). A small bullet will appear to show where the breakpoints are. When execution reaches a line with a breakpoint (but before the line is executed), the debugger window will open (if it was closed) and execution will halt. When the external is first run, the debugger window will open (assuming the global HyperDebugIt is set-up to allow it to open) as if there is a breakpoint on the first line. At this point, you would generally set the breakpoints you want in the source code listing.

You can also set breakpoints at compile time by inserting `Debug checkpoint` commands into your script. These commands are also used to invoke the HyperCard debugger when running uncompiled HyperTalk scripts under HyperCard 2.x.

You can turn off a breakpoint simply by clicking on the bullet in the margin.

- 3) You can tell what line is about to be executed by looking for the diamond marker in the left margin next to the source code. The diamond marker moves through the script denoting lines about to be executed.

The Variable Monitor

The variable monitor is located above the source code listing. It lists all variables used by the external including Toolbox globals, HyperCard globals, shared variables, locals, parameters, records, pointers, and handles. All variables are arranged by handler and show their name, data type, and value (if any).

Every handler in the monitor will list "it" as one of its variables whether or not "it" is used by that handler. When "it" is uninitialized in a handler, its data type will be a "?". This is done since "it" is used in many different ways in HyperTalk scripts so its data type may be constantly changing.

You can edit most variables simply by clicking on the value or name in the monitor. A new window will open reporting the type and, if appropriate, the size of the variable (e.g., "32-bit number" or "4-byte OType") and an editable field with the value of the variable. You can edit the value within the size limitations imposed by the data type (i.e., OTypes can only have four characters so a fifth character would be ignored). Changes you make in this editing window will take effect immediately upon clicking "OK." If the variable is a record, a hex editing window will open instead (described below). You can also open the hex editing window for any variable by Option-clicking its line. Non-printable string characters (like linefeeds or tabs) will be displayed as a small box (the standard non-printable character symbol) with the exception of carriage returns which are displayed as "↵" (option-L or option-return in HyperCard/SuperCard).

An **Edit Hex** button will be available, if appropriate, in the variable editing window as well. The **Edit Hex** button opens another window showing the actual hex values of the memory that contains the variable. If the variable is a pointer, handle, or a longInteger with a valid address in it, the memory at that location is displayed rather than the memory that contains the value of the variable.

Edit Hex is a very powerful feature for those who know what to do with it, but it is dangerous. Proceed with caution if this is unfamiliar territory to you. RAM at the location being viewed in the window is in a buffer so if you make a mistake, you can cancel the edit or revert to start over again. If you click the **Save** button, any changes you made in the window will be made directly to RAM. It is possible to crash your machine, destroy data, damage hard disks, etc., if you make a mistake.

There are cases where a variable of type `longInt` ("L") may contain a pointer or handle that DebugIt! does not know about. For this reason, if you edit one of these variables, you'll be presented with the standard variable editing window rather than the hex editing window that you might expect. DebugIt! always examines the value of these variables before displaying them and if the value makes sense as a pointer or handle, the **Edit Hex** button will also appear. If you click on the **Edit Hex** button, the hex editing window will display the memory the value points to rather than the memory that contains the value. If you know you are looking at an unlocked handle, you may wish to have DebugIt! lock the handle before displaying the memory it points to, so that the act of opening the editing window will not move the data from the locations being examined. Do this by holding down the option key while clicking the **Edit Hex** button. DebugIt! will unlock any handle it locks this way; you should not use the option key on any locked handle you want to remain locked.

While in the **Edit Hex** window, you can double-click on any hex value, and DebugIt! will convert the long integer to decimal and display the value editing dialog. If the value is a valid address, it can be dereferenced by clicking again on the **Edit Hex** button, so that you can view and edit the memory that it points too.

You can view the memory at any address, whether or not you have a pointer or handle for that address, by editing the value of "it" or "the result." "The result" is shown at the top of the variable monitor, and "it" is at the top of each handler's list of variables. Generally you would choose whichever of these values is not currently in use (i.e., is of type "?"). Just Command-click on it and enter your address, then click on the **Edit Hex** button.

If the current handler in your script has any active repeat loops, the number of repeats left will be shown at the top of the variable monitor. Just click on these values to open the variable editing window. If the loop does not have a specified number of repeats (e.g., repeat until the mouse is down), you'll have to find some other way to exit prematurely (possibly by changing the value of a variable in the variable monitor).

On the same line with "the result" in the variable monitor, the current character used to delimit items is displayed. This character is set with the ItemDelim property (not to be confused with SuperCard's ItemDel property which is different but very similar), or if the HyperCard 2 checkbox is checked, by getting or setting the ItemDelimiter property. If the ItemDelim property is set to a non-printing control character or decimal digit, then it is displayed as the (decimal) charctonum value of the character, in two digits.

The Controls

DebugIt! has a number of checkboxes and buttons. At the top left is a checkbox that lets you ignore the execution of functions and handlers called by the current script. If this is unchecked, every line that executes in Step or Trace mode is displayed; if checked, then execution runs quickly through the subroutines and returns to tracing or stepping when they return.

Moving down from the top left of the window, the first button is **Step**. The **Step** button executes the current line (where the diamond marker is) and stops before executing the next line.

The next button **Trace** executes a series of lines beginning with the current line and moving down until either all lines are executed, a breakpoint is encountered, you type Command-period, or you click on the stop sign in the upper-right corner of the window

The **Resume** button is similar to **Trace** except that no updating of the debugging window is performed (the variable monitor and current line marker will not change until you stop execution or encounter a breakpoint). This allows execution to proceed much faster but also means that you won't know what is executing. Closing the debugging window has the same effect as clicking on the **Resume** button except, of course, the debugging window is closed.

Use the **Clr Bkpts** button to clear all breakpoints in your script, including those set by the `debug checkpoint` command. Breakpoints are remembered even if you shut your machine down.

The next control **Send Msg** is sort of like the message box in HyperCard. It provides a channel of communication between the debugger and the host application. You may need to check the value in a field, set a property, navigate to some location, etc., while the external is running. (Remember, you are inside your external command at this point, so you won't be able to click in a field on a card and type a value.)

The **Send Msg** button allows you to send most HyperTalk messages to the host application. Clicking on the button will open a dialog box with a field for you to type your message into. Below the field are three buttons: **Value**, **Command**, and **Cancel**. If you type something like `the time of field 15` and then click on the **Value** button, the value will be retrieved for you. If you type `Beep` or `go card 15` and click on the **Command** button, that action will be performed.

After performing a command or requesting a value, a dialog will appear reporting whether the action was performed successfully or not. If you requested a value, the value will be the second item in this dialog. The first item in the case of a value or the only number in the case of a command will be one of the following:

0=success

1=failed

2=not implemented

It is unlikely you will ever see a 2 appear but it might if you are testing in an environment that has only limited support for the HyperCard 1.x XCMD interface. (This might happen in an environment that has no HyperTalk interpreter, like Fox Software's FoxBase which supports XCMDs in only a limited way.)

The final button is the **Refresh** button. The host application window may not be correctly updated at times (remember, you are inside your external at this point so the application is not expecting to update its screens until the external is finished). The **Refresh** button redraws the DebugIt! window and tries to ask the host application to update its windows as well. Not all host applications will respond to this request (all versions of HyperCard and SuperCard will respond, other applications may or may not). In some host applications, it may be impossible to force an update.

Above the variable monitor are two checkboxes. These are used to monitor the various callbacks. The most useful of the two is the **Review Text Callbacks** checkbox. If this is on, each time a text callback is encountered in your script, a dialog will open showing the exact text that is being sent back to the host application. This text may be somewhat different from what is shown in the source code listing. Any differences are the result of adding in the text of any variables and the addition of any string concatenation that may be required. You can cancel the callback and substitute your own value for the result, edit what is being sent back, or leave it unchanged at this point. When the host application is done executing the callback, another dialog will appear showing whether the callback was successful or not (see the result codes above under **Send Msg**) and any value that was returned. If a value is returned, you can edit it as well before executing the next line in the external.

The **Review Binary Callbacks** checkbox displays a result code after the callback is executed. If you cancel a binary callback, the executing script may become confused by the lack of any results it was expecting. Sometimes this can crash the system, so you should avoid cancelling binary callbacks except when you are prepared to deal with the consequences. DebugIt will stop execution at the end of the current line when you cancel, even if you were running in **Resume** or **Trace** mode, but it may already be too late at that time.

You'll also see a button which looks like a Stop Sign at the top-right of the debugging window. Use the stop sign to stop execution at any point. DebugIt! also checks for Command-period between lines. If you have your own trap for Command-period, it probably will not work, since DebugIt! will see it first. No matter how you resize the debugging window, the stop sign will always be positioned so that it is visible.

Some Things You Should Know About DebugIt!

It is important to note that whenever a debugging window is open, you are no longer in the world of the host application, your external has control of the machine. You cannot switch windows, perform any editing tasks in other windows, choose menu items, etc. You can switch out to other applications under MultiFinder or System 7.0, though.

If your external calls another external that also opens a debugging window, the called external will take control of the machine until it is finished at which point control will return to the first external. You cannot switch between debugging windows nor will the first debugging window be updated.

Because of the design of DebugIt!, the ROM call `FrontWindow` and anything else that calls it (e.g., the HyperCard 2.0 callback `FrontDocWindow`) will return incorrect values if any debugging windows are open. You can generally work around this by placing a breakpoint on lines that contain these calls and a second breakpoint on the line immediately following them. Close the debugging window on the first breakpoint (allowing `FrontWindow` to obtain a correct value) and the second breakpoint will cause the debugging window to open again. This technique may not work in cases where multiple debugging windows are all open at once (several externals all calling each other), in which case you may be able to edit the result using the variable monitor's editing window to correct the value.

You can drag the double-line between the variable monitor and source code listing up or down to adjust the relative size of those areas.

The added size of DebugIt! will generally not be reflected on the **Analysis** card. DebugIt! adds about 39K of code on top of your external, plus a copy of your script and the information to build the variable monitor section. The actual size of an external containing DebugIt! will vary depending on the number of variables in your script and the length of your script.

DebugIt! has been tested under HyperCard 1.1, HyperCard 1.2.5, HyperCard 2.0, HyperCard 2.0v2, HyperCard 2.1, SuperCard 1.0, and SuperCard 1.5. It is fully expected to work in other environments as well. DebugIt! is fully self-contained and does not depend on any aspect of HyperCard or SuperCard beyond the HyperCard 1.0 XCMD interface, so it should work well in any environment that supports externals.

Breakpoints, window size, window position and the split-window divider are all remembered by the window. This is true even if you quit the host application and restart. If you reopen the external in a computer with a different screen configuration than it was closed on, so that the DebugIt! window would be off-screen, it is replaced in a default position on the main screen.

If there is insufficient memory to load your external with DebugIt! attached, nothing will happen (the external will not be executed at all). The fact that the external did not execute is reported in the result of the calling script, so you'll need to design your own check for this in a way appropriate to your situation.

If you want to take advantage of ICOM Simulation's TMON debugger or Apple's MacsBug debugger to do lower level debugging chores, you can

`embed Debugger` or `DebugStr` calls in your script. Placing a dollar sign (\$) before these lines (or any other lines for that matter) instructs *CompileIt!* to compile them only if *DebugIt!* is turned on. Lines beginning with a dollar sign are considered to be comments if *DebugIt!* is not turned on.

When you have finished debugging your external, remember to recompile it with *DebugIt!* turned off. *DebugIt!* will add considerably to the execution time, memory, and size for your external so you'll want to remove the *DebugIt!* code before distributing your external.

Distributing Externals with *DebugIt!* Attached

No runtime fees or licensing are required to distribute externals created with *CompileIt!* However, if for some reason you want to keep *DebugIt!* attached to your external when you distribute it, you must include the following notice somewhere in your programs documentation:

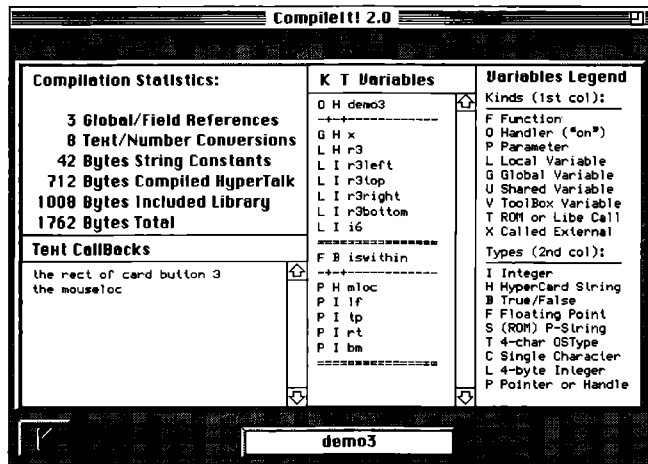
The XCMD/XFCN [external name here] includes *DebugIt!*
DebugIt! ©1991 Tom Pittman. All Rights Reserved.
DebugIt! is a trademark of Heizer Software.

No notice is required for *CompileIt!*-created externals that do not include *DebugIt!*, but a nice credit is always appreciated. If you do mention *CompileIt!* in your documentation, you should also note that *CompileIt!* is a trademark of Heizer Software in some appropriate place.

Analysis

After you've compiled a script, you can use the **Analysis** button on the **Script** card to have *CompileIt!* give you an idea of how well your script has been compiled. This button takes you to the **Analysis** card. The **Analysis** card is shown below with the analysis for the Demo3 script presented in the earlier Callbacks section of this chapter. Using the information on this card can help you create faster, more efficient externals.

The field at the upper-left corner of the card shows statistics for compilation of your external. The first line shows the number of references to global variables



or to fields in your external: the more of these references you make, the slower your compiled external will run. The performance hit is greatest when these are within a loop. The same is true of the number of text / number conversions (but less so): the more of these in your script, the slower the external will run (again, the performance hit is greatest within a loop). The third line simply tells you how many bytes in the external are strings or constants: one byte is allocated for each unique string constant byte plus a couple of bytes of overhead for length, padding, etc. Additional references to identical string constants take no additional memory.

The fourth line in this field shows the number of bytes in the external that are actually compiled HyperTalk. This is actual 68000 code, the fastest code produced by *CompileIt!* Next fastest is the Library code, which is included as part of *CompileIt!*, and glued to your external as needed.

Finally, the last field shows you the total size of your external.

The field in the lower left corner of the **Analysis** card shows you the text callbacks used by your external. Since callbacks are very slow, you can use this field to see which lines in your script produced the text callbacks. The analysis for the Demo3 external in the graphic above includes two text callbacks, for the mouseLoc and for the rect of card button 3.

The middle field gives you some details about the variables used in your external. This field has three columns: the first shows the Kind of variable, the second its Type, and the third its Name. The first line will always be the name

of your handler, preceded by either an "O" for "on handler," which creates an XCMD, or an "F," for "function," which produces an XFCN. If the script contains multiple handlers — either functions or "on" handlers — then each handler will be listed separately, together with the variables it uses, and separated from the other handlers by a double line of dashes.

Next comes the list of variables actually used by your handlers. The first column shows the *kind* of variable, as a single character; the key to these characters is shown to the right of the variables field. Toolbox procedures, functions and variables are also listed if you use them in your script.

Types of variables are a little more complex; they are shown in the second column. The first three types are fairly self explanatory: integers, HyperCard strings (in the case shown, the variable "x" as used in the Repeat Loop), and Boolean values. Booleans in HyperCard are represented by the strings "true" or "false." "F"-type variables are floating point (SANE). A legend next to the variables field shows the complete list of data types and their code letters.



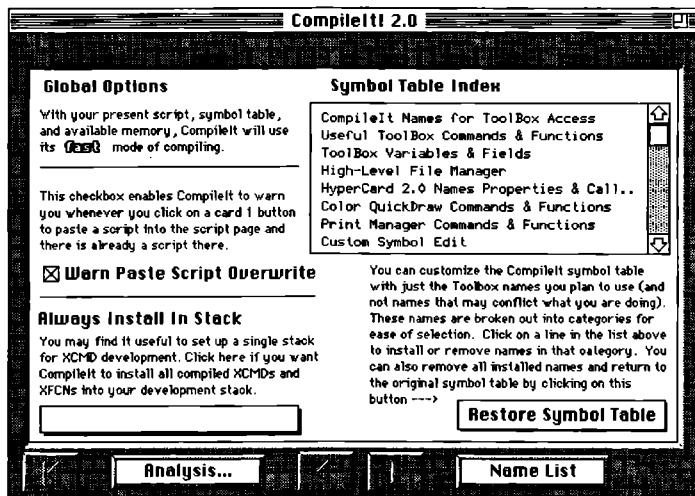
THE SYMBOL TABLE

CompileIt! works, in part, by using “lookup tables” in its translation from HyperTalk to machine language. This means that when *CompileIt!* encounters a word in one of your scripts that it does not recognize, it must look up that word in a special area called a symbol table, to find out the machine language representation of that command.

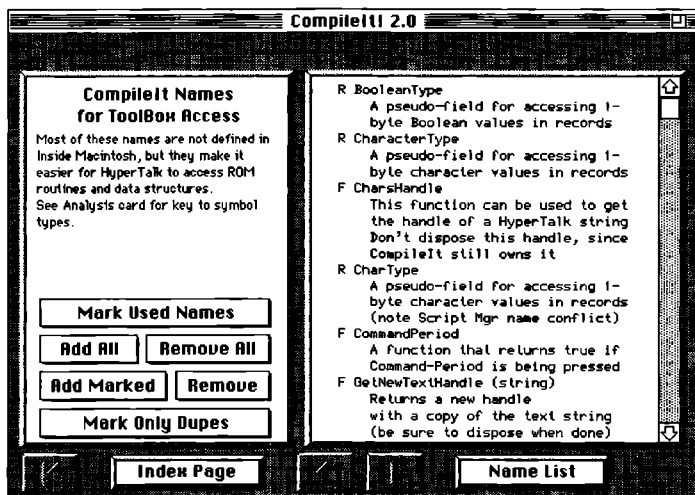
CompileIt! allows you to customize this Symbol table, depending on the type of externals you will be creating. If, for example, you are writing an external that does not use any features of the ROM Toolbox, you can make sure that these items are not installed in the Symbol table. Similarly, if you are not using any specific features of SuperCard, then these words need not be installed, either. On the other hand, if you *are* using any of these special terms, the correct symbols must be installed. The major benefit of *not* installing these words into the symbol table is that you cannot accidentally use a symbol unintentionally, thereby causing your external to fail when it runs. Another benefit is that *CompileIt!* will have a shorter symbol table to consult, so it will run much better in low memory conditions.

This modular nature of *CompileIt!*'s symbol table means that it will be possible for Heizer Software and other developers to create stacks that update the symbol table: either to modify existing symbols, or to add new symbols for such things as direct support for new programs. These modifications will likely be made available through sources such as CompuServe and America Online.

To modify *CompileIt!*'s Symbol Table, click on the **Other Options...** button on *CompileIt!*'s first card or choose “Options” from the *CompileIt!* menu. You will be taken to the card shown on the next page.



To the right of the card is a scrolling list of the different symbol sets available to you. Click on the first line in this scrolling field, and you will be taken to a card that looks something like the one shown below. Each of the cards in this Symbol Manager has the same format. The difference between the cards lies in the actual symbols managed by that card.



To add all the symbols listed in the scrolling field on this card, click on the **Add All** button; click on the **Remove All** button to remove them. After you have clicked on either one of these buttons, *CompileIt!* will work for a few seconds, adding or removing the symbols.

If you don't need *all* the Symbols listed in the field, you can select specific symbols by clicking on the ones you want in the scrolling list. As you click on a symbol, it will be marked by a bullet (•). If you click on a marked symbol, the bullet will be removed. You can add or remove marked symbols.

Next to each of the symbol names, you will see a one-character designator that *CompileIt!* uses to define the *kind* of symbol it represents. These are the most common designators you will see:

- F HyperCard or Toolbox Function or HyperCard property
- K Constant (such as the name "space" for character number 32)
- O HyperCard command or Toolbox procedure
- R Record Structure field name
- V Toolbox or user-defined variable

On the cards showing the various symbols, those taken from *Inside Macintosh* generally show the volume number and page on which that symbol is detailed in *IM*. Others derived solely from Macintosh Technical Notes are identified by the TN#. A few identify the Apple publication *Develop* as the source.

Note that the **Marked Used Names** button causes *CompileIt!* to scan the current script on the **Script** card, and mark only those names on this table that are actually used in that script. Once those symbols have been marked, the **Add Marked** button can be used to install them for use.

The following sections give you more details about some of the symbol tables you can install into, or remove from, *CompileIt!* Each of these tables is represented by its own card in *CompileIt!* From the Options screen, you can go to any of these cards by clicking in the scrolling field; when you are at any of these cards, you can use the **Next** and **Previous** buttons to get to other tables.

Note that you will have more symbol tables listed and included than are discussed here; as you will see in the *Custom Symbol Edit* section below, it is possible for libraries of symbols to be added to *CompileIt!*, either by Heizer Software or by others. Indeed, you can build your own custom symbols and save them into separate updater stacks. This is useful if you want to distribute them to others, or for updating future versions of *CompileIt!*

Also on your disk is a folder of additional symbol tables stored in updater stacks. The symbols in this folder are included for completeness, though you generally will not need them.

You will also note that the HyperTalk names are permanently installed in *CompileIt!* and cannot directly be modified. However, whenever there is a Toolbox command or function that has the same name as a HyperTalk command or function, the *Toolbox* command or function will take precedence over the HyperTalk command, except for the commands that are converted to direct 68000 machine code. Sometimes, as in the case of the `Random` function and the `Left` and `TopLeft` properties, your script may not behave as you thought it would, or you may get syntax errors. If you use these HyperTalk names, you should locate and mark them in the *Inside Macintosh* symbol lists, and use the **Remove Marked** button to remove them. Or you can click on the **Mark Only Dupes** button after installing the whole list, then **Remove Marked**.

CompileIt! will display a dialog any time you install symbols that have the same names but different meaning as other symbols already installed. The dialog will report the number of "duplicate names marked 0". These symbols will be marked by a diamond (0) in the scrolling list for quick reference. When you receive this alert, you should check to make sure these symbols will not cause any problems for you. The **Clear Marks** button at the bottom of the card will be changed to **Mark Only Dupes** so you can quickly mark all of these symbols and then click the **Remove** button to remove them.

On the **Script** card is a check box at the top of the screen labeled **Symbols**. If this option is on under HyperCard 2.0, each time you hit the return key at the end of a line, that line will be compared to the currently installed symbols and any matches will be displayed in bold type. If a script is already in the **Script** card when you turn this option on, the entire script will be compared to the symbol table. Note that this option makes NO attempt to verify syntax; it merely compares names.

***CompileIt!* Names for Toolbox Access**

These names are not necessarily defined in *Inside Macintosh*. Instead, they are used for such things as conversion between HyperCard-type data structures (i.e., text) and data structures used by the Toolbox routines. If you look at *IM*, you will see that many of the types of data used by Toolbox routines are represented in this table.

An example is the very handy `charsHandle` function. Many Toolbox routines require a handle to a block of text. Frequently your external will receive text as a parameter passed to it. If you need a handle to that text, you can get the handle using a line such as:

```
put charsHandle(theText) into theHandle
```

where `theText` is the name of the variable holding the text, and `theHandle` is the name of the variable that will hold the actual handle. You can then use `theHandle` as a parameter to any Toolbox routines that need a handle to the text they are manipulating.

If you will not be using the Macintosh Toolbox, you do not need to add these symbols to *CompileIt!*'s main table.

Useful ToolBox Commands and Functions

This symbol table includes over 500 Toolbox commands (procedures) and functions that users of *CompileIt!* are most likely to need. Again, if you are not using the Macintosh Toolbox, you do not need to install all these symbols. If you *are* using the Toolbox, you should probably just add all these symbols to *CompileIt!*'s main symbol table, unless you are operating in a 1MB Macintosh.

One of the useful features of this list is that by examining the short codes of each name, you can easily see whether you should treat it as a command or a function when using it from *CompileIt!* As mentioned earlier, procedures in *Inside Macintosh* are usually treated as if they were HyperTalk commands. These *IM* routines are marked with an "O" on this symbol list, while those that are treated as if they were HyperTalk functions are marked with an "F." Note, too, that each procedure or function generally includes a reference to the volume and page of *Inside Macintosh* that details that procedure or function.

ToolBox Variables and Fields

These are the global variables and record fields defined in *Inside Macintosh*. *IM* variables are marked with a "V" and field names are marked with an "R." For example, the variable `apFontID` contains the font number of the default application font, which affects all programs, and is thus a global variable in the Macintosh system.

Again, if you will be using *CompileIt!* a lot for compiling externals that use the Toolbox, you might as well add all these names; if you will not be using the Toolbox, you do not need them, and they actually can get in the way. Note

that there are a significant number of name conflicts between this list and HyperTalk. Be sure to review the marked duplicates carefully, and delete names you commonly use in HyperTalk. Frequent problems in compiling scripts come from the use of the HyperTalk `Value` function or `Ticks` noun with the Toolbox global variables or fields of the same name installed.

High-Level File Manager Routines

These are the high-level File Manager routines described in Volume II of *Inside Macintosh*. Though Volume II describes them well, some of these routines have been modified for use with the Hierarchical File System (HFS), so you should also review Volume IV of *Inside Macintosh*.

HyperCard 2.0 Names

Two kinds of names are on this card — new HyperTalk names introduced in HyperCard 2.0 and the names for the extended XCMD interface discussed in detail in the section *Creating xWindows* of Chapter 4.

Use of these names may limit your external to being usable only in HyperCard 2.0 and above.

SuperCard Commands and Callbacks

There are two kinds of names listed on this card. One kind of name is that used by *CompileIt!* to access those parts of SuperTalk that are entirely new to that program, or whose syntax is different from that of HyperTalk. The other kind of name listed in this table refers to those specific callbacks that are unique to SuperCard. The use of these new callbacks is described in Silicon Beach's Tech Note #6, which was included with SuperCard; most of these callbacks are used to access data stored in SuperCard's data-fork resources, or in SuperCard objects such as bitmap graphics. Note that these are the faster binary callbacks, not the slower text callbacks.

Remember that you need to include these SuperCard symbols in *CompileIt!* only if you are using SuperCard-specific commands or callbacks in your external. Most externals created using "plain vanilla" HyperTalk, including most ROM Toolbox commands and functions, will run in SuperCard as they do in HyperCard.

Color QuickDraw Commands and Functions

This symbol table contains nearly 100 Color QuickDraw-specific Toolbox utilities, from Volume 5 of *Inside Macintosh*. Of course, if you are not using color QuickDraw — even if you are using a color Mac II — you don't need to install these routines.

Additional symbol tables with some of the less-commonly used symbols are in separate updater stacks on the main *CompileIt!* disk. Look for a folder entitled "Optional Symbol Tables." The low-level file manager and many of the System 7.0 symbols are in this folder.

Custom Symbol Edit

The **Custom Symbol Edit** card allows you to define your own symbols for use in *CompileIt!* These symbols can be added to *CompileIt!*'s symbol library, and can result in faster and smaller compilation. Once you've defined your own symbols, you can use this card to create a separate stack that allows you to deliver these symbols to others.

To create a new symbol, click on the **New Name...** button on the **Custom Symbol Edit** card. The display will change, showing radio buttons indicating the different types of symbols *CompileIt!* uses. To create one of these new names, click on the radio button representing it at the left edge of the card, which defines the class of symbol you are creating. When you click on one of these buttons, another set of radio buttons appears which lets you define the type of data or return value represented by the symbol, and the name of the field in the upper right corner changes, to show you which type of data is required for that symbol. Once you've created your new symbol, you can click on the **OK** button to return you to the main custom symbol card. You can then use the **Add Marked** or **Add All** button to add this symbol to *CompileIt!*'s main symbol table.

The Symbol Editor lets you create these classes of symbols:

K Constant

You can define your own single-character constants, which will work in your compiled scripts exactly the same way that `return` and `linefeed` in HyperTalk refer to ASCII characters 13 and 10 respectively. To create,

for example, a constant called "Semicolon" that refers to a semicolon, click on the **Constant** radio button, then type "Semicolon" in the **Symbol Name** field, and "59" (the ASCII value of the semicolon character) into the **Value** field. If you want, you can type some notes about what this constant refers to in the **Code and Comments** field. Then click on the "C" data type button to tell *CompileIt!* you're creating a character constant. To create a constant string with more than one character, type the text (in a single line: returns are not allowed) into the **Value** field and choose the "S" type. Constant numbers (like pi) are similar, but you choose type "I" or "F" as appropriate.

V ROM Global Variable

Many of the ROM Global variables are already included in *CompileIt!*'s symbol libraries. However, you can define others by including their address (in decimal), and telling *CompileIt!* what kind of data is contained in the variable.

R Record Field

Again, many of *Inside Macintosh's* record structures are included with *CompileIt!* To create new fields, you need to calculate the offset of the record field from the beginning of the data that contains it, and tell *CompileIt!* that offset, as well as the type of data.

X External (XCMD/XFCN)

You can include an inline jump to another external in this custom symbol card, for faster access to other externals. Normally, when a compiled external calls a separate external, that call is handled as a text callback to HyperCard, which can be quite slow. By creating the inline jump, the second external is called directly, without passing the request to HyperCard. This means the second external is called much faster.

CompileIt! is able to recognize other externals created by it. Inline jumps to other *CompileIt!* externals will be faster than jumps to externals created by other development systems.

To create an inline jump to a second external, simply click on the button **Pre-Compiled XCMD** to the left of the card. Radio buttons will appear that let you define whether that external is a Command or a Function ("Command (no type)" and "HyperCard String Handle" respectively). All you need to do is enter the name of the external in the **Symbol Name** field and type in the number of parameters that function takes in the **# Params** field. You can include notes about what the parameters need, and the external's syntax in the **Code and Comments** field.

This only works when the external is in an open resource fork. In SuperCard, externals are installed in the data fork of the project, where they are not visible to *CompileIt!* generated code. You must either install them in the shared file as resources or open your own resource file using the *Open Resources* and *Close Resources* SuperTalk commands.

The external you install does not need to be installed in *CompileIt!* itself; it only needs to be in the stack in which you are using your compiled external. If the external is *not* in that stack (or anywhere in the hierarchy), then the value **Can't Load External** is placed into the result. You can, therefore, test to see if the jump was made correctly by calling the result function immediately after the line that calls the second external, if the external is an XCMD. If the external is an XFCN, the error message is returned as the function value.

CompileIt! does not support optional parameters (that is, where a function can take a varying number of parameters). Instead, you must pass values for each of the items in the parameter list, even though you might not need them all. You can pass empty parameters for those you do not need. Although it is probably poor form, some externals might test the paramcount to determine the number of parameters passed to them (and behave differently for different numbers of parameters). To fully utilize such externals requires a little legerdemain, mostly involving creating a different symbol for each different number of parameters.

Let's assume you want to call XCMD "Sam" with either one or three parameters. Start by creating a symbol "Sam" with three parameters, and install it in the symbol table (using **Add Marked**). Then click on the **Copy & Rename** button and select that symbol, renaming it as "Sam3". Finally **Remove All** to eliminate the original Sam from *CompileIt!*'s symbol table.

Now you can **Edit Marked** to bring up the original Sam, and change the number of parameters to one. Click OK, add it to the symbol table, and **Copy & Rename** this one as "Sam1." After removing the original Sam again, the two copies can be installed and will work correctly.

H Text Callback

Text callbacks are generally built into *CompileIt!* Creating new ones, though, allows you to extend *CompileIt!* to deal with additional callbacks included in programs that are not HyperCard compatible. Custom symbols are required only when the text callback syntax allows parameters separated by spaces and prepositions (or other words) rather than commas.

* Binary Callback

Binary callbacks are faster than text callbacks but are still performed by HyperCard. This kind of callback is used to retrieve values from fields, to access global variables, and in the case of SuperCard, to work with resources in the data fork.

To add a binary callback, you'll need to know the number and types of the inArgs and outArgs for the XCMD parameter block; *CompileIt!* will set up all the necessary glue. *CompileIt!* assumes, however, that there is at most one result value in outArgs[1], whose type is declared by choosing one of the result type radio buttons. You can have up to 8 inArgs values (the actual number is defined by Apple in the case of HyperCard, or whoever else is supplying the interface). You must type into the parameter list field the letter representing the appropriate data type. For example, if the parameters were, param1:LongInt; param2:Pointer; param3:Str255, you would type LPS or L,P,S (commas are optional).

It is important to realize that integers, Booleans, pointers, and handles are all sent as 4-byte values. Strings (both HyperCard zero-terminated and P-strings) and floating-point (extended) are passed by reference, that is, a pointer or handle is passed instead of the data itself. *CompileIt!* knows about these conventions and makes the appropriate conversions for inArgs, but callbacks generally do not pass back pointers to P-strings or extended floating point numbers allocated in their own storage, which means that there are no functions that return P-strings or floating-point values.

Unfortunately, some callbacks work as commands but are defined at the Pascal level as functions. This means that there is no outArgs value, but another inArgs is allocated for the pointer to some space that the client program has allocated for the returned value. The custom symbol edit script is not able to cope with this kind of anomaly, and if you want it to have a function interface, manual patching will be necessary. The technique for patching is beyond the scope of this manual.

\$ Raw Inline Code

This type of symbol is actual machine language that you can insert directly into your external. Adding it to a library means that you can refer to the code by name, instead of having to type the code separately each time you create an external that uses it. This discussion is beyond the scope of this manual.

GETTING TO KNOW THE TOOLBOX

This chapter provides background and tutorial material on programming with the Macintosh Toolbox. This is no easy task as the main reference, *Inside Macintosh*, is over 3000 pages. Professional Macintosh programmers also rely on hundreds of pages of technical notes, back issues of MacTutor (a great reference source), and many other references. Consider this an introduction but by no means a complete discussion of the issues involved in programming the Macintosh.

Bridging The Gap to *Inside Macintosh*

CompileIt!'s extensions for the ROM Toolbox are powerful but sometimes confusing for the uninitiated, especially since the common reference sources use Pascal rather than HyperTalk. This section summarizes information found elsewhere in this manual and presents in HyperTalk some of the more common record structures found in the ROM Toolbox.

This section should not be considered a replacement for *Inside Macintosh* nor should you consider this a complete reference. It is provided to "bridge the gap" between writing code which uses the ROM Toolbox with *CompileIt!* and the standard reference sources which generally use Pascal for examples. Consider this a "Quick Start Guide" — a more detailed discussion is presented in a later section.

In Pascal, both procedure and function calls show their parameters enclosed by parentheses. In HyperTalk, only function calls have their parameters enclosed. Below are the declarations for a function and a procedure from *Inside Macintosh* followed by the HyperTalk syntax for calling them.

A common Toolbox procedure from *Inside Macintosh*:

```
Procedure DrawString (s: Str255);
```

To express it in HyperTalk, you would write:

```
DrawString anystring
```

A common Toolbox Function from *Inside Macintosh*:

```
Function EqualPt (ptA,ptB: Point) : BOOLEAN
```

In HyperTalk, you would write:

```
put EqualPt (ptA,ptB) into x
```

Notice that the `EqualPt` function returns a Boolean value (true/false), so we used a `put` statement to express it in HyperTalk (we had to do something with the value, like put the value somewhere). When looking at a Pascal program that uses a function like `EqualPt`, you will see it expressed something like:

```
x := EqualPt (ptA, ptB)
```

which translates into the HyperTalk `put` statement shown above. You could also have just used the Boolean result directly in your HyperTalk (which looks just like the corresponding Pascal):

```
if EqualPt (ptA,ptB) then ...
```

Notice that the *Inside Macintosh* routines list both the data type for any parameters and the data type of the return value of any function. In addition to the data type information, you may also see the word `VAR` before a parameter as in this example:

```
Procedure GetIndType (VAR theType: ResType; index: INTEGER);
```

`VAR` parameters are somewhat like function return values in that a value is placed into the parameter (kind of like a two way street). Sometimes you'll need to pass a value into one of these parameters and it will be changed into a different value and other times you just need to supply an empty container to hold a new value that the routine will supply. The documentation for the routine will tell you if you need to supply a value. The `GetIndType` procedure might be used like this:

```
GetIndType theType, 1  
if theType = "ICON" then ....
```

From time to time, you may need to access a "Low-Memory Global" or "Toolbox Global." These are global variables maintained by the operating

system. *CompileIt!* makes them available for you — all you have to do is refer to them by name. (You do not have to declare them in any way, except to install them in the symbol table.) Apple warns against changing the values in these globals but it is generally safe to examine them. Low-Memory Globals of interest are listed at the end of each chapter in *Inside Macintosh*.

Notice that the data type for the two parameters for `EqualPt` is type “Point.” A Point is a **record structure**. A record structure is a collection of values stored together in one place that can be looked at as a group. The individual parts in a record structure are called **fields** (not to be confused with HyperCard fields).

Each time a new record structure is discussed in *Inside Macintosh*, its structure is shown, and also repeated at the end of the chapter. Points are defined in the QuickDraw chapter with the following definition:

```
Point = RECORD
CASE INTEGER OF
0:
    (v: INTEGER;
    h: INTEGER);
1:
    (vh: ARRAY [VHSelect] OF INTEGER);
END;
```

A Point is made up of two fields with the names “v” and “h” (vertical and horizontal). In this case, both fields are integers. (Remember our discussion of Data Types earlier in this manual — integers take up two bytes.) Two integers take up the same space as a long integer (four bytes).

Points are used to specify coordinates (screen locations) and are commonly expressed in HyperTalk as two numbers separated by a comma (e.g., `set the loc of button 1 to 100,100`).

You can create a point in one of several ways. The easiest is to use the `SetPt` procedure (*IM v1-193*) which takes two integers and combines them, returning the value in a VAR parameter.

```
SetPt myPt, 100,100
```

Or, if you don’t mind your external being compatible ONLY with HyperCard 2.0 and above, you could use one of the binary callbacks from the **HyperCard 2.0 Symbols** card as in:


```
put the loc of button 1 into xy
stringToPoint xy, myPt -- myPt is a VAR
-- myPt now has a Point record in it equal to xy
```

The reverse being:

```
PointToString myPt, xy -- xy is a VAR
set the loc of button 1 to xy
```

A similar but more complicated record structure is a Rectangle (usually abbreviated Rect) which is really two points combined and used to specify a rectangular area (e.g., put the rect of button 1). The *Inside Macintosh* definition for a Rect is:

```
Rect = RECORD
CASE INTEGER OF
0:
    (top: INTEGER;
    left: INTEGER;
    bottom: INTEGER;
    right: INTEGER);
1:
    (topLeft: Point;
    botRight: Point);
END;
```

Just like `SetPt` shown above, another procedure, `SetRect`, can be used to create a Rect. The problem is that a Rect is eight bytes, which is larger than *CompileIt!* will allow for a variable. You have to allocate some space to hold a Rect; the easiest way to do this is to use a shared variable:

```
global myRect:Record[8]

Function makeARect x, y, x1, y1
    setRect myRect, x, y, x1, y1
```

CompileIt! will make sure the memory is released automatically.

Just as with Points, there are two new callbacks in HyperCard 2.0 for creating Rects and decoding Rects back into strings that HyperCard understands (`StrToRect` and `RectToStr`).

You may want various portions of a Rect such as just the bottom value or just the right value. *Inside Macintosh* provides names for each field in a record structure to make this easy. Assuming you have a Rect in a shared variable, you access the individual fields like this:

```
put myRect.top into theTop
put myRect.bottom into theBot
put myRect.Left into theLeft
put myRect.Right into theRight
```

You may have noticed that the *Inside Macintosh* definition for Rect uses the word CASE. This means that there is more than one way to view the information in the record. Since a Rect is really two points, you may want to extract just the bottom right or top left points. Two additional fields in the Rect structure are used for this:

```
put myRect.BotRight into theBottomRight
put myRect.TopLeft into theTopLeft
```

You may have noticed by now that most of the fields in the Rect record structure have counterparts of the same name in HyperCard — the `Top`, `Bottom`, `Left`, `Right`, and `TopLeft` properties. *CompileIt!* requires that you make a decision: you can use these names only as property names or as record field names in a particular script. When you install symbols from any of the symbol cards and there are any conflicts (i.e., the `Top` record field name vs. the `Top` HyperCard property), you will be alerted that there were “duplicate symbols” — the last symbol installed in this case is the one that *CompileIt!* will use. This being the case, the following bit of code will result in an error message:

```
put myRect.Top into myTop
add the Top of field 1 to myTop
```

There are at least two ways to resolve this situation: 1) Use the **Copy & Rename** button on the **Custom Symbol Edit** card to make a copy of the conflicting symbol, and rename it. After renaming, install the copy to avoid the conflict. 2) *CompileIt!* provides a number of “pseudo” fields for the common data types (e.g., `IntegerType`, `LongIntType`, `OSType`, `SaneType`, `Str255Type`, etc.). These are all listed on the first symbols card. Since we can see from the *Inside Macintosh* definition that the `Top` field of a Rect is an integer and the first field in the record structure, we remove the `Top` record field name from the symbol table (using the buttons on the symbol table cards) and use the following code:

```
put myRect.integerType[1] into myTop
add the Top of field 1 to myTop
```

The brackets tell *CompileIt!* to extract the value of the size specified by the name of the pseudo field at the index specified inside the brackets. This is called array notation and is very useful when dealing with record structures. The following code fragments show various ways to work with a Rect using array notation:

```
put myRect.integerType[1] into myTop
put myRect.integerType[2] into myLeft
put myRect.integerType[3] into myBot
put myRect.integerType[4] into myRight

put myRect.LongIntType[1] into myTopLeft
put myRect.LongIntType[2] into myBotRight

repeat with i = 1 to 4
  put myRect.integerType[i] into item i of myRect
end repeat
```

Let's consider a practical application of the array notation for speeding up a slow script.

Consider the following pure HyperTalk script which counts the number of occurrences of a specified character in a given string:

```
function countThings string,thing
  put 0 into count
  repeat with i=1 to length(string)
    if char i of string is thing then add 1 to count
  end repeat
  return count
end countThings
```

When compiled into an XFCN, the above function is about twice as fast as the uncompiled version. The chunk expression inside the repeat loop takes most of the time. Replacing the chunk expression with some simple array notation, and performing numeric compares instead of string compares, results in a very fast XFCN — hundreds of times faster. Here is a revised version using array notation:

```

function countThings string, thing
  put 0 into count
  -- use charsHandle to get a handle to the string
  put charsHandle(string) into strHand
  put chartonum(thing) into myThing -- get the ASCII value of
thing
  repeat with i=1 to length(string)
    if chartonum(strHand@@.charType[i]) = myThing then add 1 to
count
  end repeat
  return count
end countThings

```

Notice that since we did not have a record structure to deal with, we used the `charsHandle` function to extract the handle to the string parameter. If you are working with handles (or pointers) instead of shared variables of type `Record`, you need to use the dereference operator (`@`). Use one dereference operator for pointers and two for handles.

Safe Pointers for Faster Strings

Although *CompileIt!* considerably improves the performance of ordinary HyperTalk, any Pascal or C programmer can get much better performance than that. This section introduces some of the more esoteric data type capabilities of *CompileIt!* available in Pascal or C but not in plain HyperTalk, so that you can get the same performance levels formerly possible only in those languages. You should be reasonably familiar with programming in HyperTalk before embarking on this new adventure, and it will probably help to have Volume 1 of *Inside Macintosh* handy for reference. Some of our examples also use the toolbox routines from the symbol table sections of *CompileIt!* They are not already installed in *CompileIt!* as it is shipped because there is some danger in misusing them. Back up your system (you do that regularly anyway, don't you?), install the symbols, and sally forth.

There are two things that make compiled HyperTalk scripts run slower than you might expect from machine language. One of these is text callbacks. If you have to use them, you have to use them, but at least with a little care you can move most of them out of your inner loops.

The other major slowdown is string handling, which we can improve substantially by some sharp programming. The one thing that slows string operations

down most is the whole concept of chunk expressions. Chunk expressions are a very powerful tool, but like powerful motors in 1960-era automobiles, they take a lot of gas — even if you are not using all their power. Let's begin by looking at a reasonable script to replace all occurrences of a particular character in a string by some other character:

```
-- Normal Chunk Expressions
-- replace all occurrences of aChar with bChar in string
Function ReplaceChar string, aChar, bChar
  repeat with i = 1 to length(string)
    if char i of string is aChar then put bChar into char i of
string
  end repeat
  return string
End ReplaceChar
```

You and I know that `aChar` and `bChar` are single characters, but *CompileIt!* does not know that. So it generates general code to handle all possible lengths of all three parameters. *CompileIt!* is not even smart enough to figure out the index variable `i` will never lie outside the boundary of the string, although it might actually do that if `aChar` is longer than `bChar`, which we said would not happen if this function is used correctly. So *CompileIt!* will test `i` against the current length of string on every iteration through the repeat, then compare the character it finds there against `aChar`, and if it happens to match, replace that single character with whatever string happens to be in `bChar`. That's a lot of work, but HyperCard would do as much for you also. Well, maybe a little better; they are nice folks at Apple. See below, time #3.

String compares are slow and have to be done by special routines; *CompileIt!* generally uses the Toolbox ROM to do the actual comparing, but in this case it noticed that you are comparing a single character against `aChar`, so it's a fast character compare after all. If you want a case- and diacritical-insensitive comparison, so that "a" matches "A" as well as "à" and "â" and so on, then you need to fool *CompileIt!* into thinking it's a longer string, such as:

```
if char i to i of string is aChar
```

This takes *much* longer to run, because every character comes out and gets put into its own string, then is sent off to the toolbox to compare (see time #1 below). In this case we want to replace exact copies of a particular character (perhaps a comma or tab or lower-case "e") with some other precise character (maybe a space or capital "E"). For that the machine-language byte compare is

quite adequate. *CompileIt!* will choose the optimum 68000 machine code if it knows that you know it is comparing a single character. We could also write it more precisely, though *CompileIt!* gives about the same runtime as the original script:

```
-- Character Chunk Expressions
-- replace all occurrences of aChar with bChar in string
Function ReplaceChar string, aChar, bChar
  repeat with i = 1 to length(string)
    if char i of string is first char of aChar
      then put bChar into char i of string
    end repeat
  return string
End ReplaceChar
```

The first char of aChar and char i of string are both known to be single characters (even if aChar is a longer string, its first character is still a single byte), so *CompileIt!* generates fast one-byte compares instead of the longer string comparison library calls. The put command to replace that character is not quite so simple, but the compare typically happens many more times than the put (are you replacing every single character? I didn't think so). The performance is still quite disappointing because every time you touch character i of the string, *CompileIt!* goes out and counts the number of characters in it. You never know — it might have changed. You and I know it didn't, but *CompileIt!* cannot remember whether the last time was a substitution or not, nor if the characters were replaced with the same size string.

One way to speed this up for long strings would be to eliminate the counting. Here is where some of the newer syntax helps.

Character strings in HyperTalk and *CompileIt!* are stored in memory as *handles*. A handle is a special number in the Macintosh memory called a *pointer* that points to another place in memory that contains another pointer that ultimately points to the real data. Yes, it is a little confusing; only expert programmers understand it right off, while the rest of us have to think about it a while. We'll come back to the concept of handles and pointers in more detail. For now, think of a handle as a number that is somehow connected to the place in memory where the data string is.

Normally *CompileIt!* hides the whole idea of handles from you, and you don't have to think about them at all. But now we are going to go sneaking around *CompileIt!*'s back, so we need to know what holds the strings up so we won't

trip on the guy wires. In the first symbol table card in *CompileIt!* ("CompileIt! names for Toolbox Access") there is a name you can install called `CharsHandle`. This is a pseudo-function that gives you the handle attached to any string:

```
put charshandle(mystring) into myhandle
```

With this command you can now reach in and touch the data in your string without bothering *CompileIt!* The easiest thing you can do is use something like chunk expressions, but you tell *CompileIt!* that you are looking at the characters of this handle, so it will stay away from the slow string handle manipulations, and use a toolbox routine directly to measure the handle size, which is somewhat faster than counting the characters in a long string (but probably slower for very short strings — you can't win them all). We use another pseudo symbol from the "CompileIt! names for Toolbox Access" card to let *CompileIt!* know this is a character we want to look at (and not some other kind of data). We can also use the same notation to put characters back into strings. Here is the same routine with the faster handle chunks:

```
-- Character Handle Chunks
-- replace all occurrences of aChar with bChar in string
Function ReplaceChar string, aChar, bChar
  put CharsHandle(string) into myHandle
  repeat with i = 1 to length(string)
    if char i of handle myHandle is first char of aChar
      then put bChar into char i of handle myHandle
    end repeat
  return string
End ReplaceChar
```

If you got tired of waiting for the previous examples to finish, you are in for a surprise. This one is so much faster that if you shortened the previous tests too much, there's nothing left here (see time #4).

You can also use a Pascalish array notation to get at the individual characters. Note that (unlike HyperCard chunks), character arrays start counting with 0, the way they do in *Inside Macintosh*. This is about the same speed:

```
-- Character Arrays
-- replace all occurrences of aChar with bChar in string
Function ReplaceChar string, aChar, bChar
  put CharsHandle(string) into myHandle
  repeat with i = 0 to length(string)-1
    if myHandle@@.charType[i] is first char of aChar
      then put bChar into myHandle@@.charType[i]
  end repeat
  return string
End ReplaceChar
```

Pointers and Handles

This last example introduced three new syntactical devices that Pascal programmers used to have a monopoly on. They are designated by the pointer and handle dereference operator "@", the field selection operator ".", and the array subscript designator "[...]". A serious discussion of pointers and handles is finally unavoidable.

A good time-waster in working with strings has to do with their representation in memory as handles. It's not that handles are inherently inefficient, but such things as moving them around, changing their size, and copying them take time. Professional programmers using conventional programming languages tend to allocate fixed blocks of memory for strings so that the only moving that happens is when a string is copied or characters are added or deleted. But of course they also know pretty much the maximum size their strings can grow to and they can take appropriate action if they get too big. In HyperTalk there is no way to tell the computer how big a variable will get to be, nor what to do if it gets any bigger than that. So HyperCard just allocates a variable amount of space on the heap. The heap is a large region in memory where everything variable in size or movable goes; it is shown in the MultiFinder status window as a partially-filled bar chart. If you add characters to your variable, HyperCard asks for more memory; if you delete a chunk of it, the extra space is released to be used by some other variable or resource. All HyperCard has to do is keep track of its handle.

Let's talk a little about handles and pointers. This will be a little elementary, so if you already understand pointers and handles, you might want to skim this section.

Every byte in your Macintosh memory has an address, represented by a 24-bit number. In System 7.0, Macs with a PMMU or 68030 have 32-bit addresses. All

along addresses have been stored in 32-bit (four byte) chunks of memory: the extra byte was basically unused. So let's pretend that we're "32-bit clean" as Apple puts it: all addresses take up the whole four bytes of a 32-bit word. A register in the 680x0 computer is also 32 bits wide, so it just exactly holds one address. That address "points" to some byte in your computer's memory. A number consisting of all bits zero points to address 0, which is the first location in memory. The number 00000001 points to the next byte, and so on. The first few thousand bytes in memory have been reserved to the operating system and are called "system globals." Some of those global variables have names listed in *Inside Macintosh*, like `BufPtr` and `DeskPattern`, that you can use in your compiled code (but note that the Thought Police at Apple wish you would not use them).

Many of the system globals are pointers — that is, they occupy four bytes of memory and they are themselves 32-bit numbers that contain the addresses of other places in memory. An example of a pointer is `BufPtr`. It takes up the four bytes of memory from 268-271 (hexadecimal \$010C-010F) and contains the address of the end of the unallocated memory before a program is loaded. Some INITs install themselves above `BufPtr`, moving this pointer down to reflect the reduced memory now available. An XCMD in HyperCard has little to do with this pointer, since HyperCard's CODE 0 resource and private variables are copied into the last bytes below `BufPtr`, and any change in it would probably crash either HyperCard or the whole system.

Pointers are like that: they point to a place in memory where the data is unlikely to be moved. A pointer can point to anything, since in fact any number is a pointer to something, but pointers are most useful for keeping track of specific data some place in memory. Usually a program will ask the Toolbox to allocate some previously unused memory, and return its address to you in a pointer, using the Toolbox call `NewPtr`. The system's memory manager keeps track of what memory has been allocated (and is therefore in use, showing up darkened in the MultiFinder status window) and what is still available. When the program is finished with that block of memory, it can be released for other use by calling the Toolbox routine `DisposPtr`. More on these routines later.

Let us suppose that your XCMD is going to build an index to your stack for fast searches. Suppose further that there are 120 cards in your stack, and that the index requires ten bytes per card. So you call `NewPtr`, giving it a size of 1200 bytes, and you get back a pointer to a block of memory, perhaps at address 096580. The memory manager has promised you that you own that 1200 bytes for as long as you need it, and you copy the pointer to a variable or

a field in the stack to keep it handy for future use. Now the user of your stack adds another card, so your script requires 1210 bytes for its index. You only have 1200 bytes, but you can use the system call `SetPtrSize` to increase the size of your block of memory to 1210 bytes — but only if the next ten bytes are not already allocated to some other use, perhaps some internal HyperCard function. If so, you are out of luck: `SetPtrSize` will fail. There is plenty of unused RAM available, but to take advantage of it the memory manager would have to move your data. The only trouble is that it does not know about all those copies you made of the pointer. If you tried to use one of them to access your newly-enlarged index, you would only reach the outdated old index — or worse, some data that was allocated to another request and no longer belongs to you. The memory manager will not let that happen.

The solution to being able to change the size of blocks of memory at will is to let the memory manager know about all the pointers to those movable blocks. And the easiest way to do that is to have only one pointer, called a “master pointer,” owned by the memory manager itself. All your program gets is a copy of a pointer to the pointer, called a handle. You can make copies of the handle all you want, because the memory manager promises not to move the master pointer. If the data needs to be moved to make it bigger or to make room for another memory request, the memory manager can move the data and change the number in the master pointer to reflect the new address of the data. The handle still points to the original four bytes that contain the (now modified) master pointer, and your program can reach the data at its new address as easily as before it was moved.

There is no such thing as a free lunch, right? The downside of handles is that it costs you one extra machine instruction to get to your data. Considering the other time-consuming factors in your program execution, that is not a big deal unless you are going to be touching that data thousands or millions of times. Examples of that kind of program include string copy and compare routines for very large strings. But you cannot write one of those as efficient as HyperCard’s anyway, can you? That may not be completely true, as we have seen, but it is a consideration.

As it turns out, the extra time for handle access is lost in the loop overhead. Here is our favorite example, modified to access the character data using pointer arithmetic — something that only a C hacker could appreciate. You can do it in compiled HyperTalk, but why bother? The time is slightly longer than for the array or chunk notations (see time #5), and certainly more dangerous (again like C, which is consistently slower and less robust than more advanced languages like Pascal, despite widespread prejudice to the contrary):

```

-- Pointer Arithmetic
-- replace all occurrences of aChar with bChar in string
Function ReplaceChar string, aChar, bChar
  put CharsHandle(string) into myHandle
  put myHandle@.PtrType into myPtr
  repeat length(string)
    if myPtr@.charType is first char of aChar
      then put bChar into myPtr@.charType
      add 1 to myPtr
  end repeat
  return string
End ReplaceChar

```

The Times

On a 10,000-character text file (the first three chapters of the Bible), the time in ticks it took a fairly fast MacII to replace all periods with semicolons (97 substitutions, or about 1% of the characters) for each of the algorithms in this section is:

#1. Vanilla HyperTalk, char i to i...	10722
#2. Vanilla HyperTalk, char i of...	3004
#3. HyperCard 2.1	2407
#4. Handle chunk or array notation	5
#5. C-like Pointer Arithmetic	7

Handle Chunks

CompileIt! allows chunk-like expressions on handles and pointers that you got from CharsHandle or a toolbox routine. You can think of them as somewhat like HyperTalk chunk expressions on ordinary string containers, but you insert the reserved word "Handle" or "Pointer" before the name of your container to let *CompileIt!* know that you are using the new notation. Also like HyperTalk, you can access characters, words, items, or lines — well, you can use those names, but what you really get is low-level machine chunks, which are characters (single bytes of type character), Words or Integers (two-byte numbers), or LongInts (four-byte numbers). *CompileIt!* gives you fast machine code to get to these objects. If you put something into a non-existent chunk in a handle, *CompileIt!* will extend the handle to accomodate it. Unlike

HyperTalk chunks, the inserted chunks are not empty, they are just garbage — whatever was previously there in memory. What does an empty number mean, anyway? Zero is not the same as empty, even in HyperTalk. Besides, we want this to go fast, and filling all those inserted numbers with default values would take valuable time. Pointer chunks cannot be extended, because many pointers do not refer to the beginning of a Memory Manager data block, so it cannot be resized; if you put something into a non-existent pointer chunk, Kaboom! Please be careful.

CompileIt! does not look for a delimiter between adjacent chunks in a pointer or handle structure to determine where a particular chunk is; instead, it divides all the data into equal sized chunks (you specify what type of chunk you want). This allows you to view your data in many different ways — a set of 30 integers could also be viewed as 60 characters or 15 longIntegers.

You write these special chunk expressions just as you would if you were requesting a chunk of a HyperCard field, but replace the word “field” with either “handle” or “pointer.” Here are some examples:

```
put the number of longInts in handle myHandle into x
delete first char of handle myHandle
put 150 into integer 99 of handle myHandle
repeat with i = integer 15 of handle myHandle to 255
```

A little fancier but still valid, this finds the tenth pointer in the handle myHandle, uses that as an address and finds the second integer of that structure, converts the number to a character that it stores in the third character of the structure that it located by getting the number in field ID x and converting the number to a pointer:

```
put numtochar of integer 2 of pointer 10 of handle -
  myHandle into character 3 of pointer field ID x
```

If you delete chunks of a handle, the handle will be reduced in size accordingly. If you put a chunk into a location of a handle that does not exist, the handle will be enlarged so that the chunk does exist (any chunks between the previous last chunk of the handle and the newly created chunk will contain garbage so don't rely on their being empty or zero). Note also that there is no chunk 0 in any handle chunk expressions (as there is in regular HyperCard chunks).

If you are working with large amounts of data, you may run into a situation where not enough memory is available for *CompileIt!* to perform the work a particular handle chunk expression requires. The Toolbox contains a special “low-memory global” (sometimes referred to as a “Toolbox global”) called MemErr. If *CompileIt!* tries to enlarge a handle and the attempt fails, this special global will contain a number other than 0 (probably negative). It is a good practice to check MemErr any time you think *CompileIt!* might need to enlarge a handle; if MemErr is not 0 then exit cleanly. Your code might look something like this:

```
put 10 into integer 3000 of handle myHandle
if memErr ≠ 0 then return "Error: not enough memory"
```

Pointers are a little less forgiving. If you want to use chunk expressions with pointers, note that:

1. *CompileIt!* has no way of knowing if a pointer points to the beginning of a block or at the middle of a block, so it will not enlarge a pointer to hold a nonexistent chunk. If your chunk does not exist, you will end up overwriting data that does not belong to you and will most likely crash your machine.
2. For the same reason, you cannot delete a chunk from a pointer, nor count the chunks in a pointer.

Recall that handle chunks interpret the same bits of data in the handle in whatever way your chunk calls for, with very different results. Thus if you pass a string of numbers or digits such as “1,55,33,77,99” or 155337799 to your external, HyperCard will store the data as ASCII characters, and to recover the same data you must access it as characters. You could read it as integers, but the values would be very different. The function FunnyData below will return 12588 if you call it with the string “1,55,33,77,99” since Integer 1 is the first two bytes, which are the characters “1” and “,”; interpreted as binary bits, these are 00110001 00101100, which is the decimal number 12588. Leave out the commas and it does the same interpretation on the characters “1” and “5”, giving the decimal number 12597.

```
function FunnyStuff somenumbers
  put charshandle(somenumbers) into myHandle
  return integer 1 of handle myHandle
end FunnyStuff
```

To access the numbers in a string you would need to extract them as character strings first, as in:

```
Function NotSoFunny somenumbers
  put NewHandle(0) into myArrayHand
  repeat with i=1 to the number of items in somenumbers
    put item i of somenumbers into item i of handle myArrayHand
  end repeat
  return integer 1 of handle myArrayHand
End NotSoFunny
```

Pointers and Handles Inside the Macintosh

So far we have been concerned with using pointers and handles to access data that we got from HyperCard, or constructed in our own scripts. Handles were invented for the Macintosh ROM, but occasionally some ROM routine you call wants a pointer, not a handle. That is, it expects you to give it as a parameter the address of the data, not the address of a master pointer that has the address of the data. Not generally a problem, since you can “dereference” the handle by using the “@” operator and the result is a copy of the master pointer (see the pointer arithmetic example above for an example of this). But, you recall, the memory manager knows it has the only copy of the master pointer (of course in this case that is not true). If the memory manager tries to move your data while you have the handle dereferenced, you have the same problem we discussed before, only this time the memory manager will not protect you. But there is a Toolbox call that lets the memory manager know that you are doing this and please don’t move the data for a short time. The routine is called `HLock` (short for `HandleLock`, since it locks the data down to its current address in memory). When you finish with your copy of the master pointer you must always unlock the handle using `HUnlock` so that the memory manager is free to move it around again. If you don’t unlock your handles, why bother using handles at all? Besides, it is rude when your XCMD is a guest in another program’s house (in this case HyperCard’s), not to observe a certain etiquette. HyperCard does not like locked handles laying around, and has been known to unlock them itself capriciously. Don’t tempt it.

But wait, the pointer arithmetic example at the beginning of this section did not use `HLock` — the `ReplaceChar` script is simply getting and storing individual characters, and not calling the Memory Manager for any purpose. Memory cannot move in this case so `HLock` is not needed. Many programmers believe that it is always a good idea to lock a handle before doing

anything to it. That is not true. Locking handles unnecessarily may cause your program to fail. With a few exceptions (such as `HandAndHand`, as documented in IM 1-376), you should avoid locking a handle that you are passing to the ROM without dereferencing. For example, if you lock a handle before sending it to `SetHandleSize`, the memory manager will be forbidden to move it, even if it needs more bytes than is available at the current location; the result is that you will get an "Out of memory" error while there may be still lots of empty heap space available elsewhere. It is also not always necessary to lock a handle when you dereference it, but only if you are going to call a Toolbox routine that may involve the memory manager. If you don't call any Toolbox routines yourself, and you are not manipulating HyperTalk string variables or chunk expressions (which *CompileIt!* turns into memory manager routine calls), then nothing can move your handle; locking and unlocking it can only waste time. There is a list of routines that may move memory in the back of *Inside Macintosh* Volume III (also in the back of volumes IV, V, and VI for the newer routines). While you cannot always tell which routines are called by *CompileIt!* code, you can check the line information in the DebugIt window to see if there are any "ROM/Libe" calls, which might therefore be unsafe.

You can use handles and pointers for data that you create and stuff into the freshly allocated memory, and you can use them to point to data you got from another source. There are special considerations for each kind.

Inside Mac Data Structures

If you are building new data structures in memory, you have the problem of getting the pieces of data into the right places. The pointer or master pointer only points to the beginning of the block of memory, which is not much help if you want to put data into the middle. If the data structure is defined in Inside Mac and its field names are known to *CompileIt!*, it is easy: you just use the record field notation, as for example to get at the mouse location in an event record. The words "record" and "field" refer to Pascal concepts, and have nothing at all to do with the "record" operation in a tape recorder or MacroMaker, and are only slightly related to the fields in your HyperCard stack cards and backgrounds. A record is a data structure that has component parts, which are called fields, somewhat as a card in your stack has fields. Pascal fields are accessed only by name (using that little dot), and never by number or "ID" (which means nothing in this context). Here is an example where we allocate a pointer to 16 bytes for the event record, pass it to the Toolbox routine `EventAvail`, then use the field name to access the parts:

```

put NewPtr(16) into EventRecPtr
get EventAvail(-1, EventRecPtr@)
put HiWord(EventRecPtr@.where) into mouseVert
put LoWord(EventRecPtr@.where) into mouseHorz
put EventRecPtr@.what into eventcode
DisposPtr EventRecPtr
if eventcode=1 then return mouseVert&" "&mouseHorz

```

Note that `EventAvail` (as well as `GetNextEvent` and `WaitNextEvent`) want an actual event record, not a pointer, so we dereference the pointer once. If we had used a handle, it would be necessary to lock it and dereference it twice: the first dereference gets the master pointer, and the second actually gets the record.

It is not possible for *CompileIt!* to anticipate all possible record structures, and even some from *Inside Macintosh* have been omitted. In these cases you must get at the internals of your data block the hard way. There are several ways you could do this. We already mentioned using handle or pointer chunks; there is also the Pascalish array notation, and just plain pointer arithmetic. Or you could define some new fields in the **Custom Symbol Edit** card of *CompileIt!*

First you must calculate the offsets from the beginning of the record to the pieces you want, to get to those parts. Recall that a pointer is just a number that happens to be the address of some bytes in memory. The event record, for example, is 16 bytes long. When we call `NewPtr`, it returns the address of the first of those sixteen bytes. To reach the `where` field (if it were not defined in *CompileIt!*'s symbol table), you would have to add up the sizes of all the pieces between the front of the record and that field. In this case there are ten bytes in front of it, so the offset is 10. The `what` field is the first, so its offset is zero; we can use the original pointer. The same example, using only pointer arithmetic and not field names looks like this:

```

put NewPtr(16) into EventRecPtr
get EventAvail(-1, EventRecPtr@)
put EventRecPtr+10 into wherebyte
put HiWord(wherebyte@.longIntType) into mouseVert
put LoWord(wherebyte@.longIntType) into mouseHorz
put EventRecPtr@.integertype into eventcode
DisposPtr EventRecPtr
if eventcode=1 then return mouseVert&" "&mouseHorz

```


Note that we are still using record notation here, but with the pseudo fields, `longIntType` and `integertype` to get the right size of data. If you don't use some kind of field name, *CompileIt!* will not know what kind of data (byte? 16-bit? 32-bit?) to get from memory, and your results will probably not come out right — if it compiles at all. Since `where` is defined as a `Point`, which is two 2-byte numbers, we could also have written:

```
put wherebyte@.integerType into mouseVert
```

but that would involve calculating a second offset for the horizontal part of the mouse location:

```
add 2 to wherebyte
put wherebyte@.integerType into mouseHorz
```

When the chunk offset is an exact multiple of the chunk size, we can also use the array notation:

```
put EventRecPtr@.integerType[6] into mouseVert
put EventRecPtr@.integerType[7] into mouseHorz
```

Note that an offset of 10 bytes is only 5 two-byte integers. Since the integer arrays start counting with 1 instead of zero, byte offset 10 is integer number 6:

Byte Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Integer #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
LongInt #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

As we can see, it is not possible to get to offset 10 by subscripting `LongInts`, since 10 is not an exact multiple of 4. So you could not use array notation to reach the `where` field of an event record. We leave as an exercise to do the same thing using pointer chunks.

Note also — and this is important — that although we were willing to add 2 to `wherebyte`, we did not add 10 to `EventRecPtr`. We could have, but then it would not point to the front of the block in memory. If you give a pointer to `DisposPtr` (or any other memory manager routine) that does point to the beginning of its block, you will confuse it so bad that a bomb is inevitable. You can make all the copies of a pointer you want, with whatever offsets added to

them, but be sure to save an original (unmodified pointer or copy) for sending to the memory manager routines. This applies also to `SetPtrSize`, `GetPtrSize`, and the corresponding handle manipulation routines.

Now, where did the offset 10 come from? In *Inside Macintosh*, Volume I, page 264 (IM I-264 for short), there is a description of the `EventRecord`. The first offset is always zero. Then you add the size of that field to get the next offset, and so on. If a field is itself a record, then you must add up the sizes of all the fields in the inner record to get its size. Note that the offset for everything except 1-byte components must be even, so add one if necessary to take the offset up to an even number. Generally, *Inside Macintosh* does not throw you any curves like that. The size of each of the most common data types follows:

1	Boolean
1	SignedByte
2	Char (actually only one byte, but Apple's compiler allocates two unless it's "packed", so the rest of us have to go along with that particularity)
2	Integer
4	LongInt
4	Fixed
4	Ptr (or any other pointer)
4	Handle (of any kind)
4	Point
4	OSType
8	Rect
8	Pattern
12	"SANE" extended floating point number (really 10, but <i>CompileIt!</i> allows for 12 in its code)

CompileIt! includes special names (accessible from the **Options** card) that make it easy for you to access these data types.

Arrays in *Inside Macintosh* structures are slightly more complex. You calculate the size of the component, add 1 if necessary to make it even (unless it is just one byte and the array is "PACKED"), then multiply by the number of elements. Sets allocate only one bit for each element, so count the number of elements, divide by 8, and round up to the next even number of bytes. You should now be able to calculate, as an exercise, the size and offsets for a `BitMap` (IM I-202). Then try the `GrafPort` on IM I-203, and see if you get 108 bytes.

Once you understand how to calculate the offsets to the internal parts of an Inside Mac data structure, you can easily add any names you like to the *CompileIt!* symbol table. You still have to calculate their offsets correctly, but once you do that, they go into the symbol table and you can safely forget about them.

More About String Handles

All HyperCard variables are accessed by handles. *CompileIt!* knows this and hides the details from you. You refer to a variable *x* as in:

```
put bg field 3 into x
```

and *CompileIt!* will ask HyperCard for a copy of the string that contains the data in the field. HyperCard will return a handle, and *CompileIt!* will make sure that your XCMD takes good care of that handle. If you make a copy of the variable in HyperTalk, for example:

```
put x into y
```

CompileIt! will make a copy of the handle — not a copy of the whole string, so it is not unnecessarily copying large numbers of characters — and put a copy of the handle in the variable called *x* also into the variable called *y*. Suppose field 3 has the string “The quick brown fox” and that *x* now contains a handle that points to a copy of that string in memory. When we put *x* into *y*, *y* now has a copy of the same handle, pointing to the same string (in the same bytes in memory). Now it gets tricky. Let’s say you change the value in *y*:

```
delete second word of y
```

What happened to *x*? Answer: nothing. *CompileIt!* noticed that you were calling for *y* to be modified, and saw that there was another handle pointing to the same string (meaning some variable that obviously should not be modified at the same time), so it makes a copy at that time, and modifies the new copy, leaving the new handle in *y*. Variable *x* still has the handle to the old (unmodified) string. You can watch this happen in DebugIt! Type the following short script into *CompileIt!* and compile it with DebugIt! checked on:

```

on LookAtHandles
  put "The quick brown fox" into x
  put x into y
  debug checkpoint -- stop here and look
  delete second word of y
  debug checkpoint -- look again
end LookAtHandles

```

Run this XCMD until you hit the first breakpoint, then option-click on the *y* variable line. The first eight bytes of the hex display are the handles respectively in *y* and *x*; they are identical because they point to the same physical string. Step over to the next breakpoint and look again. Now *y* has a different handle in it.

There is a little bit of overhead in keeping track of which variables have what handles, but for strings longer than a hundred characters, the savings by copying only handles, not strings, should pay off. But all this involves no effort at all on your part. You don't even need to be reading this to take advantage of string handles this way.

The problem is that when you ask for a chunk expression of any kind, *CompileIt!* cannot give you the original string with the excess characters removed if there are other variables holding handles to the same string. So it makes a copy of the string, cut down to size. The only exception is if you ask for a one-character chunk and do not put it into a variable. Then *CompileIt!* will just grab the single character. Here are several ways to access the same single character in a string, and the time (in seconds) it takes to do it 100,000 times on a Mac IIci with a 16-character string in *x*:

(A) 366

```

put char i of x into y
if y="A" then donothing

```

(B) 1,166

```

put char i to i of x into y
if y="A" then donothing

```

(C) 1.5

```

if char i of x = y then donothing

```

(D) 1.5

```
if char i of x = "A" then donothing
```

(E) 1.5

```
if CharToNum(char i of x) = 65 then donothing
```

Examples (F) to (H) all assume the following line is just before the timing loop:

```
put CharsHandle(x) into hx
```

(F) 0.25, almost as fast as empty loop!

```
put hx@.ptrtype+i-1 into h  
if CharToNum(h@.chartype) = 65 then donothing
```

(G) 0.25

```
if char i of handle hx = "A" then donothing
```

(H) 0.25

```
if hx@@.chartype[i-1] = "A" then donothing
```

(I) 0.25

```
if char i of handle hx = y then donothing
```

Example (A) extracts a character from the string then converts the isolated character into a string to store in string variable *y*. This conversion requires the additional handle management for the new string in *y*. Example (B) makes a new handle and then copies the selected character chunk into it. It just happens that only one character is in the selected chunk, but *CompileIt!* assumes from the form that there may be several characters, so a more general form is used.

The remaining examples use the character as extracted, without storing it into a string variable. No conversion is needed, and they are quite fast compared to (A) and (B). Comparing the character to a constant is about the same speed as comparing it to a variable or running it through `CharToNum` to change it to a number. Note that (A) and (B) are true HyperCard case-insensitive compares using the Toolbox international string compare function, while (C)-(I) compare for exact match using a few machine instructions. That is some of the reason for the extra speed.

What we have done in (F) is taken a copy of the handle to the string, then used a pointer scheme to access the characters. While every HyperTalk string is maintained as a handle to a block of memory, *CompileIt!* hides this representation from the casual observer. Other handles (not HyperCard strings) can be processed using the Toolbox routines and pointer access, and the programmer is responsible for knowing what he is doing. `CharsHandle` is a data structure defined in *Inside Macintosh* as a handle to a block of characters — exactly a HyperCard string — but *CompileIt!* will not do the usual handle management on it. The first line of this example sneaks a copy of a string variable handle out to let you do ordinary handle operations on it. Making this copy is a little costly in time, so you should make only one copy and use it for all your manipulations. Note also that although this is a proper handle, you should not dispose it because *CompileIt!* still is using its own copies of the same handle. You can do some other tricks, however, as we shall see. Dereference the handle once (one “@”) and you have a pointer to the first character in the string. You can add a character offset to that pointer to point to any character in the string, or else use the array notation (H) for the same effect. Note again that in true computer style, the first offset is 0; HyperCard sometimes spoils us with human counting systems that start at 1. That means that to access the same character as the other examples, we must subtract 1 from the offset. This is also true of the array subscripting form (H), but only in the case of characters (following the lead of *Inside Mac's* `CharsHandle` data type); other array types follow the Pascal (and HyperCard) conventions of numbering from 1. Using the pseudo-type `chartype` in both cases, we get one byte from the string. There is no checking for a legitimate offset, so if you run off the end of the string you get garbage, not empty as in the other examples. This also makes the previous three examples run a little slower, since they must count the number of characters in the string to see if `i` is greater or less. That takes time. If you know how big the string is (one call to the function `length` is sufficient), you can do your own checking once instead of on every character access.

All that may be all well and good for comparing characters, but what about doing something with the characters? If you want to operate on individual

characters, your best bet is to extract them as in example (E) or (F), converting them immediately to numbers by means of `CharToNum`, then work with the numbers after that. Numbers are extremely fast in the Macintosh CPU; character strings are rather slow. If you want a string of several characters, you might find it faster to examine the characters one by one to determine the starting and ending offsets, then use one chunk expression to extract that substring all at once. Thus you are only making one additional copy of the string.

If you are working with small chunks of a very large string variable, you would like to avoid *CompileIt!*'s nasty habit of examining the whole string to take a chunk out. There is a way, using Toolbox routines `BlockMove` and `Munger`. There are two situations where this can help: the most important is extracting a chunk with the least amount of character shuffling, but you can also insert or delete a few characters without making a whole copy.

Let's say you have a handle `hand` for a large block of text in memory. You have cleverly passed only the handle (as a number) back and forth from your stack script to the XCMD, so the characters themselves are never touched unnecessarily. Now you want an XFCN to return the *n*th line from this giant string. Here's how it might work:

```
function getnthline hand,n
  put hand+0 into ht -- convert to number only once, here
  put n+0 into nth -- ditto
  Hlock ht -- not for search, but needed before copy
  -- first find the nth line by counting return characters
  put ht@.ptrtype into pt -- use this to step thru array,
  repeat while nth>1 -- looking for nth line...
    put CharToNum(pt@.chartype) into thechar -- is a number!
    if thechar=0 then -- end: there is no nth line
      HUnlock ht -- don't forget this
      return empty
    else if thechar=13 then subtract 1 from nth --13=return
    add 1 to pt
  end repeat
  put pt into start -- save start of nth line
  -- now count the number of characters in that line
  put CharToNum(pt@.chartype) into thechar
  repeat while thechar>13 -- look for end of line or string
    add 1 to pt
    put CharToNum(pt@.chartype) into thechar
  end repeat
```

```

-- copy the line out into HyperCard-style string
put pt-start into thesize -- = length of the line
put "anything" into astring -- create a string variable
put CharsHandle(astring) into strhand -- get its handle
SetHandleSize strhand,thesize+1 -- 1 byte for null at end
HUnlock ht -- safe now: BlockMove does not move handles
if thesize>0 -- don't bother if no bytes to move
then BlockMove start,strhand@,thesize -- copy line to var
put NumToChar(0) into strhand@.chartype[thesize] -- null
return astring -- with new size and new data in it!
end getnthline

```

There are several points that you should pay attention to. First, after deciding how many characters to copy into the return string, we must be careful to put a null character at the end. HyperCard and *CompileIt!* normally do that automatically for us with ordinary strings, but this is behind their backs. Second, the string variable `astring` had to be initialized with something to tell *CompileIt!* that it is a text variable (not a number or whatever). Use a string literal, not another string variable, because that text will be replaced when *CompileIt!* is not looking. Finally, when we create the string variable `astring`, and when we change its size by `SetHandleSize`, we implicitly or explicitly call on the ROM memory manager to do these things. This can move other handles around if necessary to make room, so if `ht` is not locked it would invalidate our pointers `pt` and `start`. That's why we lock down the handle. If we create the string `astring` and set its size to the correct value before getting a pointer value `pt`, then locking would be unnecessary. `BlockMove` is not a memory manager call, and it does not reposition handles.

Of course this technique can be adapted to any kind of text (or even non-text) data. Note that at the end of this function we had three or four old pointers laying around. That's OK, since the only data of interest is in the handle that we carefully locked and unlocked, and in the return string. Extra handles and pointers don't hurt anything as long as you don't try to use them after they become obsolete. This is true only if the pointers and handles are derived from data owned by some other program.

Who Owns This Handle

You should think of each handle as being "owned" by whatever program is responsible for the data. In the case of resources from a file (any program or file resources), the owner is the Resource Manager: it is responsible for allocat-

ing the space, and disposing of the handles when the file is closed. Strings that came as parameters to your XCMD are owned by HyperCard, and it is responsible for disposing of them when your XCMD returns. HyperCard also takes ownership of the string you return in the return command, and it owns some of the strings passed to and from external windows. *CompileIt!* (or rather the XCMD it compiled) owns the strings you use as variables in your XCMD. It also owns the copies of fields and global variables it gets from HyperCard. You only own the handles you create by `NewHandle` or its equivalent. When you call `GetResource` or another Toolbox routine that returns a handle to a resource, you can make copies of the handle, and modify the data, and do basically anything you like, but you must remember that the Resource Manager owns the data and all handles pointing to it are only “loaned” to you. Do not dispose of these handles. Similarly, when you use `CharsHandle` to get a handle to a HyperTalk string owned by HyperCard or *CompileIt!*, you can modify the data, you can make copies of the handle and even index into the data by adding numbers to copies of the master pointer, but *CompileIt!* will assume it owns the data and it will dispose of the handle when your XCMD exits, or when it has reason to believe the data is no longer being used.

How does ownership of a handle change? There are several ways. If you call `DetachResource` with the handle of a resource, the Resource Manager relinquishes ownership of the handle. You now own it, and you are responsible for disposing of it. On the other hand, calling `AddResource` sells a handle you own to the Resource Manager, and you no longer own it. Obviously you cannot call `AddResource` with a handle you got from `GetResource`, because the Resource Manager already owns that handle — unless you call `DetachResource` first. Similarly, you cannot honestly call `AddResource` with a handle you got from `CharsHandle`, because it is not yours to sell. *CompileIt!* will fight the Resource Manager for it, and you may not get the result you wanted (even if you don’t get a bomb). If you have a handle in `x` that you do not own, you can get a handle to a copy of the data in a handle you do own by calling `HandToHand` with the handle `x`. If you already own the handle `x` and call `HandToHand`, then you will own two handles, but `x` will only have the new one. If you did not save a copy of `x` in some other variable, that handle will be an orphan with no way to properly dispose it, left rotting in the heap and taking up valuable space until you quit from HyperCard. You can give a handle back to *CompileIt!* with a call to `HyperCardText`. You can make *CompileIt!* give up ownership of a string handle by calling `HyperCardOwns` on the string itself (not the handle). If HyperCard really owns the handle (as for example in the case of strings sent to or from an external window that does not use *CompileIt!*’s access code), then you have succeeded in stopping *CompileIt!* from fighting over it; if

HyperCard did not own the handle, then you stole it fair and square from *CompileIt!* (so you now own it). Another way you can get data into strings that *CompileIt!* owns is to create a handle by putting a quoted string into a new variable in HyperTalk, then borrow the handle (using `CharsHandle` to get it) long enough to copy your own data into it, as we did in the example above. You can `SetHandleSize`, `Munger`, and `BlockMove` data into anybody's handles — within reason, of course.

What about changing the data of a handle? If you have two handles, `x` and `y`, that some other program owns, and you want to move the data from one to the other, how do you do that? For example, let's say *CompileIt!* owns `x` and the Resource Manager owns `y`:

```
put field "x" into varx
put CharsHandle(varx) into x
put GetResource("TEXT",1000) into y
```

You would like to change the data in the resource to be a copy of the data in the field. Here are some plausible, but wrong, ways to try to do it:

```
put x into y -- y is now a copy of the
-- handle owned by CompileIt!
-- the resource is unchanged, and
-- you have no handle to it.
put x@ into y@ -- NO! NO! NO!
-- Never change a master pointer (bomb)
put x@@ into y@@ -- Will change a few bytes in
-- the resource, but not the whole resource.
BlockMove x, y, length(varx) --
-- Bomb: changing master pointers again!
-- BlockMove wants pointers, not handles.
BlockMove x@, y@, length(x) -- good start, but y
-- is the wrong size, and the length of x will
-- be 6 or 7 (number of digits in a handle)
```

The right way, of course, is:

```
SetHandleSize y, length(varx)+1 --
-- the extra byte for a null at end
BlockMove x@, y@, length(varx)+1 --
-- no need to lock handles
```

It is often a requirement to put something into a chunk of a larger string. When *CompileIt!* sees that you are changing a chunk in a string, it will (if necessary) make a working copy of the string, then call the Toolbox routine `Munger` to do the insertion. If you know you don't need to keep the original string around, you could call `Munger` yourself, and the result will be a little over twice as fast. Of course for short strings the cost of making an extra copy is small, but if you are working with 20,000 characters it could make a difference of several seconds. Here is an example:

```
-- ticks for 1000 times on Mac IIci:
put x into char i to k of y
put charshandle(x) into hx
put charshandle(y) into hy

-- 25 ticks for 1000 times:
HLock hx -- required for hx (not hy)
-- since Munger calls memory manager
get Munger(hy,i-1,nil,k-i+1,hx@.ptrtype, length(x))
HUnlock hx

...
-- 56 ticks
put c into char i of y -- (c is one character var)
-- 43 ticks:
put "A" into char i of y
-- 0.1 ticks:
put hx+i-1 into h
put "A" into h@.chartype
```

Notice that putting a single character into a 1-character chunk is slightly faster than putting in a string that will change the size of the result (but about the same speed as replacing a chunk with another the same size); a character constant is even faster. Using pointer arithmetic and the `chartype` pseudo-field (or handle chunks, or array notation, both of which are about the same speed) is virtually instantaneous by comparison.

It is easy to see that touching characters takes time — the more you touch, the longer it takes. Every time you do something with a chunk expression, *CompileIt!* goes out and counts the number of characters (or words or lines or whatever) in the string, which touches every character once. It will usually also make a copy of the string, which touches each character a couple more times. Doing your own string management will save a lot of this touching, because you can eliminate unnecessary counting and copying that *CompileIt!*

cannot safely omit. Of course it is not necessary to go to all this trouble unless you are touching the same string thousands of times, such as within a repeat loop. Note that if you were writing your XCMD in C you would have to do all string manipulations the hard way, not just the ones in the inner loops.

Toolbox Calls with Text

The Macintosh ROM was designed to be used with Apple Pascal, so almost all text information is passed as so-called "P-strings" consisting of a one-byte length code, followed by up to 255 bytes of character data. Most of the Toolbox calls take string parameters in such a way that *CompileIt!* knows that is happening and can do the necessary conversions. There are some exceptions, like when a string is part of a larger data structure such as the title in a ControlRecord (IM I-335). If you had to build this record in memory from scratch you would use the Str255type pseudo-field and pointer arithmetic to get at it.

Although the ControlRecord fields are already in the *CompileIt!* symbol table, there are some situations where there are no defined record field names. An example is resource types like the 'STR ' and 'STR#' resources. The 'STR ' resource is rather simple, since it is just a handle to a block of memory that contains exactly one P-string. If you are going to create one of these, you must be sure to allocate the right number of bytes, but *CompileIt!* will copy them in or out for you correctly:

```
on NewSTR thetext,theID,thename -- error checks omitted
  put NewHandle(length(theText)+1) into thehandle
  put thetext into thehandle@@.Str255type
  AddResource thehandle,"STR ",theID,thename
end NewSTR
```

It's much trickier to do 'STR#' strings correctly — in fact they cannot be done at all in standard Apple Pascal, which is why there is a [not in ROM] Toolbox call to get them for you (see IM I-468). Here is how you would do the same thing with *CompileIt!*:

```

function GetIndString strListID,index -- returns theString
  put GetResource("STR#",strListID) into ahandle
  if ahandle=nil then return empty
  put ahandle@.ptrtype into aptr
  put index-1 into counter
  if counter>=aptr@.integertype or counter<0 then return empty
  add 2 to aptr
  HLock ahandle
  repeat while counter>0
    add CharToNum(aptr@.chartype)+1 to aptr
    subtract 1 from counter
  end repeat
  put aptr@.Str255type into theString
  HUnlock ahandle
  ReleaseResource ahandle -- not necessary unless memory is tight
  return theString
end GetIndString

```

We leave as an exercise for the reader a script to construct a 'STR#' resource from a variable (typically a parameter to the XCMD) containing a list of strings, each on a separate line. Hint: the resource is 3 bytes longer than the length of your HyperCard variable (assuming no return at the end of the last line). Why?

Shared Variables

Once you understand the toolbox data structures, it is an easy step to telling *CompileIt!* about your own variables of a particular type. *CompileIt!* does not allow you to do this for ordinary local variables, but there is a special class of variables that are not quite global (though we use the same word) and not quite local (though they are as fast to access as local variables). Since these variables are visible to and shared by all the handlers in a single compiled script much as global variables can be shared by all handlers in all scripts, we call them "shared variables." You can just name them on a global command line before your regular handlers (but in the script being compiled), and *CompileIt!* will assign to them a data type that it determines from their usage.

The real power of shared variables comes from the ability to tell *CompileIt!* explicitly what data type they should be. This is particularly useful for making small record structures (Events, Rects, BitMaps, file system Parameter Blocks,

etc.), and for forcing temporary variables to be type Character (one byte, not a string!) or P-string (for repeated use with toolbox routines without conversion or memory manager calls). The types you can use are anything that CompileIt knows about (see the list below). With records and P-strings (type R and S) you can also specify how many bytes they should be. Here are some examples of shared variable declarations:

```
global myString:S[64] -- a short P-string
global aRect:R[8] -- a rectangle
global onech:Char -- a single character
global manyInts:R[1024] -- a record full of integers
```

In the last example, you would access the individual integers of this record using array notation:

```
put anumber into manyInts.integertype[i]
```

Note that accessing a shared variable record involves no pointers, so no dereferencing operator is required (nor should it be used, unless you have an array of pointers, or some such). You cannot use array notation to subscript variable-sized structures like Records and P-strings; *CompileIt!* cannot figure out easily how to index them. One-, two-, and four-byte data can be subscripted, though.

Allocating a pointer using `NewPtr` may require thousands or even (in a worst case) millions of bytes to be moved to make room for it. Allocating a handle using `NewHandle` typically does not move as many bytes of the heap, but if you must lock it for very long, it could lead to heap fragmentation and eventual program failure. Allocating space in shared variables costs almost nothing, and you don't have to dispose it when you are done. Shared variables are allocated on the CPU stack, and they go away automatically when your external returns to HyperCard.

Apple recommends that applications allow 8K for desk accessories, so you should try to keep your shared variable space down to 2K or less. However, if there is sufficient memory available, *CompileIt!* can handle up to 30K total shared variables. If there is not sufficient memory, you will get a bomb, ID=28. Unfortunately the space is allocated before you can test for the space, so it's too late to prevent it. Be conservative, and allocate your big data in handles on the heap, where you can control it.

Shared variables can be any of the following types:

I,L,P	4-byte integer or pointer or handle
S	Str255 P-string, 256 bytes, or whatever size you specify
H	HyperCard String Handle (the default if you don't specify a type)
O,T	4-byte OType string
B	1-byte Boolean (true/false value) allocated 4 bytes
C	1-byte Character (much faster than local string variables)
F	12-byte SANE number
R	32-byte record (unless you specify a different size)

The syntax for shared variables is the same as for HyperCard global variables with two optional additions:

1. If the name is followed by a colon (":"), then the next non-space character is used to specify the shared variable's data type.
2. If the shared variable is type Record, then its size is set to 32-bytes unless it is followed by two brackets around a number; the number is the size in bytes that will be allocated (e.g., `aRect:Record[8]`); similarly, you can specify the length of a S-type string if you want it less than 256 bytes.

Array Notation

Array notation eliminates the need for most pointer arithmetic (you'll still need it from time to time). Array notation allows you to quickly access any portion of a record provided the data in the record is one of the fixed-length data types *CompileIt!* knows about. Consider the common requirement of `SFGetFile` for an array of 4 OTypes (each OType contains the 4 character code used to identify a type of file, `SFGetFile` uses this array to filter out unwanted file types from its display). Here is how you would create such an array using array notation and shared variables:

```
global typeList:Record[16] -- 4 OTypes x 4 bytes
```

```
function GetAFile ftype1,ftype2,ftype3,ftype4
  put ftype1 into typeList.OSType[1]
  put ftype2 into typeList.OSType[2]
  put ftype3 into typeList.OSType[3]
  put ftype4 into typeList.OSType[4]
  -- etc.
```

You could also write:

```
global typeList:Record[16]
```

```
function GetAFile
  repeat with i = 1 to 4
    put param(i) into typeList.OSType[i]
  end repeat
  -- etc.
```

or:

```
global typeList:Record[16]
```

```
function GetAFile paramtypes
  repeat with i = 1 to 4
    put item i of paramtypes into typeList.OSType[i]
  end repeat
  -- etc.
```

You can mix and match quite a bit with record fields and arrays, as we have already seen in the examples above.



CREATING XWINDOIDS

This chapter discusses how to write code for xWindoids. xWindoids are XCMDs that create and “own” a window. HyperCard automatically tells xWindoids about various system events that apply to them such as mouseDowns, updates, menubar clicks, etc. Examples of xWindoids are the variable watcher and script editor. xWindoids are a new feature of HyperCard 2.0 and any XCMDs that you create using the techniques in this section will be incompatible with earlier versions of HyperCard.

This chapter is for advanced users of *CompileIt!* and assumes basic knowledge of pointers and handles, the ROM Toolbox, and the concept of event-driven programming.

It is assumed that you have the *HyperCard 2.0 Script Language Guide* or similar reference which documents the “HyperCard 2.0 Extended XCMD Interface.” This section documents only the *CompileIt!* implementation of the new interface. A summary of the entire extended XCMD interface is located in the appendices.

CompileIt! can be used to create for HyperCard 2.0 xWindoids (external windows) that respond to events, support reentrancy, and have properties that you can get or set from scripts. *CompileIt!* simplifies the code you need to write to support an xWindoid by taking its cue from HyperCard and treating events like system messages (like the mouseUp handler you might write for a button).

In other development systems, you are generally required to check for a negative parameter count and then determine what kind of event HyperCard is sending you. Once you know what kind of event is happening in your window, you have to retrieve the needed parameters and branch to a routine that deals with the event.

CompileIt! simplifies this process by providing a series of reserved handler names. All you have to do is include handlers for the specific events you wish to deal with. *CompileIt!* will automatically ensure that those routines are called directly with the needed parameters when those events occur, and ensure that events you do not specifically wish to deal with are handled in a default way.

The structure of an xWindoid script is simple. The first handler is like any other external and is used to create the window or palette using some HyperCard-2.0 specific callbacks and normal Toolbox Window Manager calls. Below the first handler are handlers, named with reserved *CompileIt!* event handler names, which deal with the specific events.

The reserved event handler names are:

```
on InitializeMessageWatcher
on InitializeVariableWatcher
on InitializeScriptEditor theScript,windowNamePtr,TalkObjectPtr
on InitializeDebugger theScript,windowNamePtr,TalkObjectPtr

on idleEvent eventInfoPtr
on mouseDownEvent eventInfoPtr
on keyDownEvent eventInfoPtr
on autoKeyEvent eventInfoPtr
on updateEvent eventInfoPtr
on activateEvent eventInfoPtr
on app4Event eventInfoPtr

on OpenEvent eventInfoPtr
on GiveUpEditEvent eventInfoPtr
on HidePalettesEvent eventInfoPtr
on ShowPalettesEvent eventInfoPtr
on EditEvent eventInfoPtr
on SendEvent eventInfoPtr
on MenuEvent eventInfoPtr
on MBarClickedEvent eventInfoPtr

on ShowWatchInfoEvent eventInfoPtr
on ScriptErrorEvent eventInfoPtr
on DebugErrorEvent eventInfoPtr
on DebugStepEvent eventInfoPtr
on DebugTraceEvent eventInfoPtr
on DebugFinishedEvent eventInfoPtr

function GiveUpSoundEvent eventInfoPtr -- return true if ok
function CursorWithinEvent eventInfoPtr -- return true for HC to
set cursor
```

In order for *CompileIt!* to recognize that your script will be an xWindoid, you must follow a few rules in how you use the reserved event handler names:

1. You must use the handler names as shown above.
2. You must use the exact same number of parameters as shown above. The parameters can have any name you wish as long as the number of parameters is the same.
3. You must NOT call any of the reserved handlers from other handlers in your script, although reserved handlers can call other non-reserved handlers.

After a compile, any event handler that *CompileIt!* recognized will be listed at the bottom of the analysis card.

Below is a simple xWindow shell showing the structure for a simple xWindow. The events SetPropEvent, GetPropEvent, EditEvent, updateEvent, OpenEvent, and CloseEvent all require special handling, so they are shown with sample code in the shell.

```
-- xWindow source code shell for CompileIt!

on myXwindow
  -- This is where you would put your code to create the xWindow
  -- Normally, you'll use either the GetNewXWindow or NewXWindow
  -- callbacks to create your xWindow.
end myXwindow

-- this handler is called when the user types
-- "set <property> of window <windowName> to <propertyValue>"
function SetPropEvent eventInfoPtr, propNamePtr, propertyValue
  if propNamePtr@.str255type = "whatever" then
    get propertyValue -- and do something with it
  else return true -- reject any other properties
  return false
end SetPropEvent

-- this handler is called when the user types
-- "get <property> of window <windowName>"
function GetPropEvent eventInfoPtr, propNamePtr
  -- return property value or exit if none
  if propNamePtr@.str255type = "whatever" then return "itsvalue"
  else exit GetPropEvent -- reject any other properties
end GetPropEvent
```

```

-- cut, copy, clear, or paste event while window has edit
on EditEvent eventInfoPtr
    if eventInfoPtr@.what=xEditUndo then --undo event
    else if eventInfoPtr@.what=xEditCut then -- cut event
    else if eventInfoPtr@.what=xEditCopy then -- copy event
    else if eventInfoPtr@.what=xEditPaste then -- paste event
    else if eventInfoPtr@.what=xEditClear then -- clear event
end EditEvent

-- your window needs to be redrawn for some reason
on updateEvent eventInfoPtr
    GetPort myWindow
    BeginUpdate myWindow
    -- redraw your window here
    EndUpdate myWindow
end updateEvent

-- your window just opened, this is the first event your window
-- will receive
on OpenEvent eventInfoPtr
    -- if you want any null events, include the following line:
    SetXWIdleTime eventInfoPtr@.eventWindow, someTicks
    -- someTicks is a number.
    -- to support reentrant code (good idea), include this line:
    XWAllowReEntrancy eventInfoPtr@.eventWindow, true, true
    -- This next line is generally a good idea:
    XWAlwaysMoveHigh eventInfoPtr@.eventWindow, true
    -- any other setup code should go here
end OpenEvent

-- HyperCard is requesting permission to close your window,
-- HyperCard cannot quit if you return false.
function CloseEvent eventInfoPtr
    -- put any code here to save data, etc. before closing
    return true -- if ok for HyperCard to close your window
end CloseEvent

```

The source code for a simple xWindoid with several properties is on the **Examples** card in the *CompileIt!* stack.

For reference, here is the structure of the eventInfo parameter block. The eventInfoPtr parameter, shown above, points to this block:

```
EventInfoPtr = RECORD
  event: EventRecord;
  eventWindow: WindowPtr;
  eventParams: ARRAY[1..9] OF LongInt;
  eventResult: Handle;
END;
```

To get, for example, the second element of the eventParams array, just write:

```
put EventInfoPtr@.eventParams.LongIntType[2] into myVar
```

You should use either the `NewXWindow` or `GetNewXWindow` callbacks to create your xWindooids. Always use the `CloseXWindow` callback to close your xWindooid (this will cause HyperCard to issue a `CloseEvent` message for your window). Never use any of the following Toolbox routines to close an xWindooid: `CloseWindow`, `DisposeWindow`, `CloseDialog`, or `DisposDialog`.

What Else You Should Know About the XCMD Interface

In addition to the way *CompileIt!* handles the creation of xWindooids, there are several other differences you should know about *CompileIt!* and the HyperCard 2.0 XCMD interface. The callbacks that are different or not needed are discussed below.

In the Script Language Guide, the first parameter to each callback is shown as `paramPtr`, because the Pascal .o file does not have ready access to that pointer. *CompileIt!* knows about this pointer already, so it is completely unnecessary.

There are a few other cases where HyperTalk as compiled by *CompileIt!* is somewhat simpler than Pascal. These are shown below. Note that there are more symbols for HyperCard 2.0 than are listed here. Several *CompileIt!* - specific symbols are documented below, as they are also of use in working with the XCMD Interface.

BoolToStr

You get the same effect whenever you put a true or false value (perhaps the result of a comparison) into a string variable. Instead of:

BoolToStr theboolean, thestring

just write (you get exactly the same effect, but faster):

put theboolean into thestring

entryPoint

This is the address used by the callback glue to jump back into HyperCard. There is nothing useful you can do with it, since invoking any callback by name automatically jumps to that address.

EvalExpr

This callback is used automatically by *CompileIt!* whenever you use the function `value(...)`.

eventResult

This record field is of value only if you are creating xWindoids in a non-standard way (i.e., you are not using the reserved handler names).

EventResult is defined as a HyperCard ("H") zero-terminated string value, to be used when you get a `xGetPropEvt` event. If you don't do anything special to the value you put there, *CompileIt!* will dutifully dispose the string before returning, which will confuse HyperCard no end (and probably bomb). The command `HyperCardOwns` (documented below) will properly instruct *CompileIt!* to relinquish ownership of the string handle so it works correctly. Be sure to use `HyperCardOwns` on all xWindoid property values, both when you get them from HyperCard, and when you send them back. (*CompileIt!* does this automatically if you use the reserved handler names.)

ExtToStr

This callback is used automatically by *CompileIt!* whenever you put a SANE value into a string variable.

GetFieldByID

GetFieldByName

GetFieldByNum

SetFieldByID

SetFieldByName

SetFieldByNum

One of these callbacks is used automatically by *CompileIt!* whenever you refer to a field by ID, name, or number.

GetFieldTE

SetFieldTE

Apple documents these callbacks as requiring a non-zero integer as an ID or field number, or else a non-nil string pointer. Since HyperCard apparently checks them in left-to-right order, you can just pass it any name (or "") for a name when you wish to refer to a field by ID or number. *CompileIt!* thus expects to see a name or "" for the last parameter, and not a string pointer or nil.

GetVarInfo

Apple gives you both a handle and a Str255 copy of the variable text (assuming it will fit). In *CompileIt!*, these are merged into one parameter, so all you have to do is put a local variable name after the `isGlobal` parameter, and that variable will receive the requested value. Be sure to initialize the variable by putting something into it first:

```
put " " into theValue
GetVarInfo handlernum, varnum, varname, isGlobal, theValue
```

HyperCardOwns

Sometimes HyperCard sends an external window a zero-terminated string handle and expects the external to dispose of it properly when it finishes with it; other times HyperCard plans to dispose of it. In the Good Old Days this distinction was fairly clear: parameters and XFCN return values were disposed of by HyperCard, and everything else (i.e., the handles used in callbacks) were explicitly the responsibility of the external. *CompileIt!* knew about this and did the Right Thing. All that has changed with HyperCard 2.0. If you use the reserved handler names in your xWindoids, then *CompileIt!* will take care of this for you, otherwise you must handle the situation. Use `HyperCardOwns` on each xWindow property value you get from or send to HyperCard (because HyperCard will be disposing of those handles), and not otherwise.

The `HyperCardOwns` command is also useful in another, more common situation. Suppose you are creating an xWindoid that will accept some data in parameters when it is first called and you want that data to stay around across events. The recommended place to store data across events is in your

window's RefCon field (actually you would store a handle to your data in the RefCon). The following script fragment illustrates the technique:

```
global arect:Record[8] -- temporary rect structure

On MyWindow X
  put X into data -- convert param into local
  put charsHandle(data) into dataHandle -- get handle to local
  HyperCardOwns dataHandle -- assume responsibility for the handle
  setRect arect,100,100,300,300
  -- create the window
  get NewXWindow(arect,"Untitled",true,0,false,false)
  if it is not nil then selectWindow it -- activate it
  SetWRefCon it,dataHandle -- store datahandle in the RefCon
  field
  -- of the window record
end MyWindow

on OpenEvent eventInfoPtr -- first event the window receives
  SetXWIdleTime eventInfoPtr@.eventWindow, 0
  XWAllowReEntrancy eventInfoPtr@.eventWindow, true, true
  XWAlwaysMoveHigh eventInfoPtr@.eventWindow, true
  -- retrieve the handle
  put GetWRefCon(eventInfoPtr@.eventWindow) into dataHandle
  -- do something with the data in datahandle
end OpenEvent
```

HyperCardText

This function tells *CompileIt!* that the handle you give it as a parameter is really a string, so it takes ownership and treats the handle as an H-type string like any other string value. If you also send the resulting string variable (before doing anything to it) to HyperCardOwns, then *CompileIt!* will be careful not to dispose it when your XCMD exits.

paramCountX

Because HyperCard has otherwise defined the function, paramCount, this field is spelled slightly differently. Note that *CompileIt!* will give you the contents of this field anyway if you use the HyperTalk function paramCount.

PasToZero

You get the same effect whenever you put a Str255 value into a string variable.

PrintTEHandle

It's unnecessary and tedious to construct a string pointer for the header string in this callback. Just give it the string directly:

```
PrintTEHandle hTE, "Some header this is"&&pageno
```

resultX

The `result` has a special meaning in HyperTalk, so this field has been renamed. Ordinarily you only need to look at this field for function callbacks, since command callbacks give you its value in HyperTalk's `the result`.

ReturnToPas

You can get the same effect by writing:

```
put first line of whatever into mystring
```

returnValue

This field is where an external puts its return value when it returns to HyperCard. If you put something there, *CompileIt!* will smash it with its own return value (whatever expression value you give the final `return` command, or else `nil` if you don't exit with a `return`).

ScanToZero

You can get the same effect by writing:

```
get the length of whatever
```

SendCardMessage

This callback is used automatically by *CompileIt!* whenever you use the `do` command.

GetGlobal

SetGlobal

These callbacks are used automatically by *CompileIt!* whenever you use a HyperTalk global variable in your script.

StringEqual

You can get the same effect by writing:

```
... something = somethingelse ...
```

StringLength

You can get the same effect by writing:

```
... the length of whatever ...
```

StringMatch

This callback is used automatically by *CompileIt!* whenever you use the `offset` function.

StrToBool

You get the same effect as this callback whenever you use a parameter or global or string variable as a boolean.

StrToExt

This callback is used automatically by *CompileIt!* whenever you use a parameter or global or string variable as a number that must be converted to SANE.

StrToLong

StrToNum

You can get the same effect as these callbacks (and probably slightly faster) if you just write:

```
put stringValue+0 into thenumber
```

GetXCmdPtr

This function returns the `XCmdPtr`.

ZeroToPas

You get the same effect as this callback whenever you put a string value into a `Str255` field or toolbox global or ROM call parameter.



APPENDIX A

COMPILEIT! AND OTHER APPLICATIONS

A number of software manufacturers have incorporated into their programs the ability to use externals created for HyperCard. Other manufacturers have extended the abilities of their externals to go beyond the abilities given to externals by HyperCard. Since *CompileIt!* is a tool for creating externals, it is not so closely tied to HyperCard that you are limited to creating externals for HyperCard with this product. Indeed, externals created by *CompileIt!* can be used in any product that accepts HyperCard externals. There are, however, a few things to keep in mind when writing externals for use with other programs:

1. Be careful not to use stack- or HyperCard-specific callbacks. If you are writing externals for use in non-HyperCard type programs, a number of the callbacks can fail. Don't refer to such things as cards, backgrounds, and stacks. Fields and buttons might give you some problems too.
2. Some environments may include built-in functions that can take optional parameters. These will not be compiled correctly with *CompileIt!* You can sometimes avoid problems in these cases by supplying dummy parameters such as `empty`, or by creating separate symbols for the different numbers of parameters.
3. Externals compiled with the **HyperCard 2** option may require HyperCard 2.0 or later. Currently, `Exit to HyperCard` and `ItemDelimiter` are the only features this option enables. Future versions of *CompileIt!* may enable additional symbols that will result in HyperCard specific code.
4. `xWindoids` require the HyperCard 2.0 Extended XCMD Interface. Many programs that support externals do not support this extended interface.

DebugIt! is not tied to the HyperCard application nor is it an `xWindoid` so it should work in many environments beyond HyperCard. Specific problems to watch for are documented in the *Debugging* section of Chapter 3.

SuperCard

SuperCard is generally very amenable to HyperCard externals, and most that we have tested work fine. SuperCard features an enhanced set of callbacks, giving you access to SuperCard features not available in HyperCard. These enhanced callbacks are documented in SBS Tech Note #6, which was included with SuperCard. If you wish to use these new callbacks, or to compile SuperTalk scripts that use SuperCard-specific commands (such as `Paste` or `SetWindow`), you will need to use the **Other Options** facility of *CompileIt!* to install SuperCard support into *CompileIt!*'s Symbol table.

CompileIt! also works as a SuperCard project. See the appendix *SuperCard and CompileIt!* for instructions on how to convert *CompileIt!* and other comments.

APPENDIX B

SUPERCARD AND *COMPILEIT!*

This appendix explains how to use *CompileIt!* with Silicon Beach/Aldus's SuperCard, including how to convert *CompileIt!* into a SuperCard project.

CompileIt! supports SuperCard 1.5. Future and previous versions of SuperCard may have more and/or different limitations than those discussed below. Also, the process of converting *CompileIt!* into a SuperCard project, as described below, may be different for versions of SuperCard greater than 1.6. Although *CompileIt!* will compile under SuperCard 1.0 once properly converted, some of the scripts that depend on SuperCard 1.5 features may need modification to complete the conversion. We assume that all SuperCard users have already upgraded to the latest version.

Whether or not you choose to convert *CompileIt!* into a SuperCard project, most externals that you create can be imported into SuperCard via the **Import Resources** command in SuperEdit. You also do not need to convert *CompileIt!* to use any of the SuperCard specific symbols (commands, callbacks, properties, etc.) although externals using these symbols may only work when imported into SuperCard.

The process of converting *CompileIt!* into a SuperCard project is simple. The steps are:

1. Make sure you have a backup copy of the *CompileIt!* disk.
2. Start with a fresh copy of *CompileIt!* in HyperCard 1.2 format. (*CompileIt!* is shipped in this format. SuperEdit may fail if you try to convert a HyperCard 2.0 stack.) You may want to place *CompileIt!* in a new folder so that it is easy to find when it comes time to locate it from SuperEdit (SuperEdit will display all HyperCard stacks regardless of format).
3. Open SuperEdit, click Cancel in the initial **Get File** dialog that appears, and choose **Convert Stack...** from the **File** menu. Select *CompileIt!*
4. When asked by SuperEdit, assign a name for the converted version of *CompileIt!* (we recommend "CompileIt! 2.0.SC").

5. SuperEdit will display a dialog asking for the type of resource conversion you want performed. Select **Custom** and in the dialog that appears, uncheck the checkbox in the bottom right. SuperEdit will begin the conversion process which takes 3-5 minutes on a Macintosh II class machine.
6. When the conversion is completed, choose **Run** from the **File** menu (or type Command-R). Do NOT make any changes to *CompileIt!* at this point.
7. *CompileIt!* will open in SuperCard and sense that it has just been converted. It will ask you to locate a HyperCard version of itself (best to select the same copy of *CompileIt!* that you selected to start the conversion process in the first place). *CompileIt!* will complete the conversion process for you (shuffling appropriate resources into the data fork, changing some of its icons, and various other "tweaks"). When it finishes, you'll be able to begin compiling scripts. You may want to go to the SuperCard symbols card (available via the **Other Options...** button) and install any or all of the SuperCard symbols at this point. If you want to make further modifications to the new SuperCard project, it is now safe to do so (at least insofar as not interfering with the conversion process).

When converted to a SuperCard project, *CompileIt!* will automatically install any externals you create into the data fork of the projects you select. This is the correct thing to do under SuperCard (under HyperCard, resources are stored in the resource fork of a stack).

If for some reason you want *CompileIt!* to install into the resource fork of a project, you'll need to modify the window script. Some reasons why you might want to do this are: 1) you want an external installed in the sharedfile, or 2) your external will be called directly by another external (see **Custom Symbol Edit**) rather than from a script. To force *CompileIt!* to install into the resource fork of a project, locate the line in the "Prelude" function in the Window script that begins with:

```
if superc then  
  if myname is not empty then put "::go"&&quote&myname...
```

Remove (or comment out) the line with the two colons. The colons tell *CompileIt!* to install into the data fork. If they are gone, *CompileIt!* uses the same installation routines that it would use in HyperCard. Remember to put the line back if you want subsequent compiles to install in the data fork again.

If you do this very often, you might consider modifying the script to look at a checkbox that lets you choose the destination without re-opening the window script each time. Much of *CompileIt!* is intentionally left as scripts so that you can customize it to your own requirements.

Several SuperTalk commands and functions take optional parameters (e.g., `Offset` and `LineOffset`). Optional parameters are generally supported by *CompileIt!* only in text callbacks that are not installed in the symbol table. Note that `LineOffset` compiles to a text callback, so it can be used only with global variables or fields (because it only makes sense with data that contains multiple lines, and returns cannot be passed in text callback parameters). `Offset` is a *CompileIt!* intrinsic and only works with the optional parameter form if you install it from the SuperCard Names card.

The **HyperCard 2** option on the first card in *CompileIt!* has no effect in SuperCard, and is grayed out.

`Exit to SuperCard` and `Exit to HyperCard` cannot be used in SuperCard.

Properties of SuperCard such as `WordDel` and `ItemDel` do not have any effect INSIDE an external although they will have an effect in certain callbacks (mostly text callbacks). This is also true under HyperCard but presents much less of a problem since most of HyperCard's global properties do not directly affect data.

CompileIt! does have one global property of its own (`ItemDelim`) which closely parallels one of the most common SuperCard global properties. `ItemDelim` allows you to change the delimiter used to separate items, to some single character value that you specify (the default value is a comma). The `ItemDelim` property *only* affects data *inside* an external and has no effect on callbacks. You can set the *CompileIt!* `ItemDelim` property to match SuperCard's `ItemDel` property in either direction:

```
set the itemDelim to the ItemDel -- set CompileIt!'s itemDel
set the itemDel to the itemDelim -- set SuperCard's itemDel
```

You should also be aware that you cannot refer to objects that are not on the current card. Statements like `put 10 into field 1 of card 10 of window 5` will generate a compiler error.

1

APPENDIX C

COMPILED VS. UNCOMPILED HYPERTALK

This appendix discusses aspects of compiled HyperTalk that are different from uncompiled HyperTalk, and covers pitfalls you may encounter.

Mod and Div — *CompileIt!* uses integer `mod` and `div` operators which are slightly different from HyperCard's equivalent operators. Consider the following statement:

```
put 1.5 mod 1 into x
```

HyperCard would put 0.5 into `x` while *CompileIt!* would put 0 into `x`. If this matters to you, you can extract the fractional part of a real number using the following technique:

```
put 1.5 - trunc(1.5) into x
```

Params, Param, ParamCount — These three functions return parameter information for the current handler in HyperCard. Compiled, they return parameter information for the whole external regardless of which handler they appear in within it.

If your external performs a direct call to another external (whose name was added to the symbol table with Custom Edit) then these functions in the called external *may* return parameter information for the external that was called from HyperCard (the first external in the chain). This condition applies if the called external was compiled by *CompileIt!* and does not use shared variables. (*CompileIt!* will not allow sharing between externals.)

Answer, Ask, Read, Convert — These four commands are text callbacks and perform as expected with one slight difference: they modify the value of `it` in the calling handler. If you need to preserve the value in `it` across calls to externals that use any of these commands, you can incorporate the `SaveIt` handler shown on the Examples card in *CompileIt!* into your externals source code. `SaveIt` is part of the "Rename" example which shows it how it should be used.

Text Callbacks — Text callbacks cannot contain return characters. Consider the following statement:

```
set the script of button 1 to x
```

where *x* is a local variable that contains a script. This statement produces no result because the `Set` command is a text callback and *x* contains a return character. Under these circumstances, the callback would fail.

The best way to understand this is to view your external as existing in a separate world from HyperCard. Callbacks are a gateway through which your external can communicate with HyperCard. HyperCard imposes certain limitations on this communication gateway so that it can quickly interpret and respond. These limitations are:

1. Callbacks must be all on one line (no return characters).
2. The total length of the callback, including both command and data, must not exceed 255 characters (a P-string).

Global variables are special in that they can exist in *both* worlds at the same time. If a particular callback in your external must be larger than 255 characters and/or contain return characters, use a global variable instead of a local variable.

Min, Max, and Average — HyperCard will accept a maximum of only 64 parameters for these functions. *CompileIt!* sets no such limit so you can pass any number of parameters to these functions.

Do — In HyperCard, the `Do` command applies to the current handler. In an external, the `Do` command applies to HyperCard, not to your external. Consider the following script fragment:

```
repeat with i = 1 to 10  
  do "put 10+" &i &&"into var"&i  
end repeat
```

In HyperCard, the `Do` statement above would create 10 new local variables named `var1-` `var10`, each initialized with a different value. Each of these new local variables would be available immediately to the current script. This statement would be meaningless in compiled HyperTalk because the `Do` statement would cause the variables to be created in HyperCard rather than

inside the external — the variables would be created in a kind of never-never land where you have no access. The only way such a statement would work would be if you already had *global* variables with the same names in both your external and in the HyperCard script that invokes it.

Global variables — If you use global variables in your externals, they must be declared *both* in the script for the external and in the script that calls the external. HyperCard version 1.2x is an exception to this rule.

Accessing objects not on the current card — HyperCard does not allow externals to access fields outside the current card, except by text callbacks (which *CompileIt!* will not generate automatically for that purpose). The following statement works fine uncompiled but will generate a compiler error if you try to compile it:

```
put x into field 1 of card 10
```

The work-around is to make the card that contains the object the current card or to let HyperCard do the work for you using the `Do` command:

```
go to card 10  
put x into field 1
```

```
do "put"&&quote&x&quote&" into field 1 of card 10"
```

Case-Sensitive Compares — All string comparisons in HyperCard are case-insensitive (e.g., "a" = "A"). This is also true in compiled HyperTalk except for single-character compares. Case-sensitive comparisons result in much faster code. The following are considered single characters and will be case-sensitive in compares:

1. A chunk expression referring to a single character (and not a range of characters that happens to be of length 1), as in:

```
char n of x -- but not: char n to n of x
```

2. A HyperCard or Toolbox function that returns a single character:

```
NumToChar(65)
```

3. A variable passed to a VAR parameter of type CHAR in a Toolbox routine; markChar is a known character in the script containing this Toolbox call:

```
GetItemMark myMenu,itm,markChar
```

4. A shared variable of type Character:

```
global myChar:Character
```

5. A string of length one, only if compared whole for equality to a known character (and not merely to another string variable), as is y in:

```
if char 1 of x = y then ...
```

Chunk expressions are generally considered to be ordinary strings, and would normally compare case-insensitive:

```
put "abc" into x
put "ABC" into y
if char 1 to 1 of x = char 1 to 1 of y then... -- a = A would be true
```

Comparing a whole string to a known character will also be case-insensitive if you use an ordering compare such as > or ≤. These operators do not affect the comparison of two known characters, which will still be case-sensitive. Putting a single character into a variable is not enough to convince *CompileIt!* that it is “a known character” on a subsequent line. (*CompileIt!* does not remember much from line to line.)

ItemDelimiter and ItemDelim — HyperCard 2.1 introduces a new property called *ItemDelimiter* that allows you to change the delimiter character used to separate items in a string. Normally, the item delimiter is a comma as it has always been in the past. By setting the *itemDelimiter* property to some other character (e.g., set the *itemDelimiter* to colon) you can change how items are viewed in a string.

CompileIt! has special support for this property. If a script is compiled with the HyperCard 2 option turned on then the following conditions apply:

1. Each time your external is called, the current value of the `itemDelimiter` property will be retrieved from HyperCard and will apply to strings inside your external.
2. When you set or get the `itemDelimiter` property inside your external, *CompileIt!* will respect the (possibly new) current setting.
3. *CompileIt!* will NOT notice if the `itemDelimiter` property changes outside your external (perhaps a text callback invokes an uncompiled handler that changes it or you do a direct call to another external that changes it), unless and until you do another `get the itemDelimiter` or otherwise explicitly look at it again.

If you are not running under HyperCard 2.1 or you compile your script with the **HyperCard 2** option turned off, then `itemDelimiter` is treated just like any other text callback and has no effect inside your external. All is not lost, though, because *CompileIt!* includes its own similar property called `itemDelim`, which works regardless of the external environment.

`ItemDelim` is just like `ItemDelimiter` except that it ONLY applies to compiled HyperTalk. It is not tied to any specific version of HyperCard. In *DebugIt!*, the current `itemDelim` character is shown in the variable monitor at the far left side of the line listing the `result`.



7

APPENDIX D

ERROR MESSAGES

If *CompileIt!* reaches a point where it cannot make sense of a script, it will stop with a “boing” and issue an error message. The message is inserted as a comment line after the line currently being scanned on the script page, and a dialog reports the next symbol on that line that has not yet been fully processed. The line where the error is detected will be highlighted. Often the real error occurred some time before it was detected, or the error message may appear irrelevant to the real problem. This is typical with most programming language compilers. The possible error messages are listed here, with some suggestions for curing each problem.

<symbol> is not a command

CompileIt! will not let you use a variable name or function name as if it were a command. Neither will HyperCard, but who asked. Perhaps you misspelled a word? Check the installed names in the **Name List** card to be sure you are not trying to use a ToolBox global or field name as a handler name in your script.

'<symbol>' is not a variable name

The quoted symbol or name is in the symbol table or your script, but it is not defined as a variable. It might be a Toolbox name, a constant, a record field, or perhaps a handler in your own script. *CompileIt!* wants to see a variable here.

'=' expected in repeat with

CompileIt! expects that the next symbol after the variable name in a repeat with construct should be an equal symbol (followed by an initial value expression, etc.).

After 'add', 'to' is expected

After 'subtract', 'from' is expected

After 'multiply', 'by' is expected

After 'divide', 'by' is expected

CompileIt! is expecting the appropriate preposition separating the two

operands of this command. Perhaps there is something wrong with the first operand, and *CompileIt!* gave up too soon.

Callbacks not allowed here

CompileIt! has a special flag to tell it that callbacks are not allowed. You should not be using that flag unless you know what you are doing. If you did not intend for this to happen, then perhaps your copy of *CompileIt!* has been corrupted, and you need to restore it from the master disk.

Can't access data structure here

Records are not allowed by *CompileIt!* among your local variables. Use a shared variable or a data structure on the heap. You also cannot use structured variable components within a text callback. Access the component in a separate line, and put its value into a local variable, then use the local variable in the callback command or function.

Can't access field that way

HyperCard has not told *CompileIt!* how many fields there are, so we cannot ask for the `middle` or `last` (or `any`) field. Please use a name, number, or ID.

Can't add that

Can't subtract that

Can't multiply that

Can't divide that

Can't delete that

These commands all act on whole containers or chunk expressions that are part of a container. You can't operate on an expression that is made up of constants, function values, and/or parts of them — where would the result go?

Can't find a repeat for this 'next'

CompileIt! found a `next` command that was not followed by the word `repeat`.

Can't get that address in A0

You can use the `""` operator in `INLINE` commands only for local variables and handlers. Global variables, fields, and callbacks do not have an address available.

Can't pass local variable to structure

Some ROM Toolbox routines require data structures greater than four bytes for parameters; *CompileIt!* cannot allow these to be local variables, since there is no way to allocate the correct amount of memory for them. You should find out from *Inside Macintosh* how many bytes are required for this structure, then use `NewPtr` or `NewHandle` to allocate the space, or else declare a shared variable of type `Record` the right size. If you use a pointer or handle, be sure to dereference the pointer with `@`, or the handle with `@@`, when you pass it to the Toolbox routine. If you are using a handle, be careful that the Toolbox routine you are calling does not call the memory manager, or, if it does, that your handle is locked before passing it dereferenced. Better still, don't use handles for this.

Can't put there

The most likely cause for this error message is if you try to put something into a chunk expression built up from a function or constant value (instead of a container). Even HyperCard won't let you do that.

Can't put there yet

Chunk expressions are not allowed on the heap. A future version of *CompileIt!* may allow this.

Can't tell if field access by name or number

This probably won't happen to you, since *CompileIt!* now calls a library routine to try to figure out whether the variable you gave it is a number or a name. But we left the error message in just for old times' sake.

Can't understand where to put

You can put something into a variable or field or a chunk of a variable or field, before or after a variable or field or chunk, or you can just leave the destination off entirely and let HyperCard put the value into the message box. Any other destination will not be understood by *CompileIt!*

One possible cause for this error is if the expression to be put into the destination is poorly written, so that *CompileIt!* assumes it has reached the end of the expression while there is still some left.

Cannot exit <symbol> from here

CompileIt! found an `exit` command that does not correspond to the current handler name nor a repeat loop of any kind.

Chunk expression expected

If you use one of the ordinal numbers (`first`, `third`, etc.), then *CompileIt!* expects you to follow it with a chunk expression.

Command or structure name expected

Every command must begin with a word that starts with some alphabetic letter. Anything else is obviously an error.

Debugger name conflict

CompileIt! knows about `debug checkpoint` and `debug coderesource` commands, but not any other command that begins with the word `debug`.

End of handler expected

CompileIt! can't find the end of your handler. Perhaps the word `end` is missing.

End of line expected

Except for if-then-else constructs, HyperTalk allows only one command on each line. If the command appears to end before the end of the line, this message will notify you that something went wrong on this line (though it is likely that the error occurred much earlier than where it was detected).

End of repeat expected

After compiling the commands inside a repeat loop, *CompileIt!* expects to find an `end repeat` line. If it does not, it probably means that the current line makes no sense as a command; this probably has nothing to do with the enclosing repeat loop at all.

Exit to HyperCard only works in HyperCard 2

CompileIt! has no way to tell earlier versions of HyperCard to stop running scripts. If you really want your external to be able to use this command, check the **Hypercard 2** checkbox on the title card.

Expected comma or ‘)’ here

CompileIt! thinks it got to the end of a parameter in this function or command call, but the next character does not make sense from that perspective. Perhaps there is something wrong with the expression here?

Expected the word ‘to’ here

It does not make much sense to convert a container without telling HyperCard what format to convert it to — perhaps you are using an improper chunk expression that *CompileIt!* choked on.

Expected the word ‘in’

To count the number of chunks, *CompileIt!* has to have you specify where the chunks are. You do that by saying the number of <chunks> in whatever.

Expecting the word ‘field’ here

HyperCard only gives XCMDs access to fields, not buttons or pictures, nor cards or backgrounds themselves.

Expression value expected

This is the generic error for poorly written expressions.

Function parameter list must begin with ‘(’

CompileIt! recognizes the identifier immediately to the left of this symbol as a function name (either built-in, or declared in this script, or else in the ROM Toolbox). A proper function call has parentheses around the parameters, or if there are no parameters, then empty parentheses. If you did not intend this to be a function call, you might consider choosing a different name to avoid the name clash.

Hexadecimal constants expected for `INLINE`

The `INLINE` command parameters are not of the right form.

Hexadecimal constants must be 16 bits or less

`INLINE` can only generate 16-bit words. If you need a 32-bit constant, break it into two 16-bit pieces.

I don't know what a `<symbol>` is

CompileIt! knows how to ask HyperCard (through a callback) whether your expression is a number, integer, point, rect, date, or logical. It looks like you have a different question to ask. Try forming your own callback with the `value` function.

Improper 'end if'

CompileIt! found the keyword `end` at the end of an if-then-end-if or if-then-else-end-if construct, but it is missing the word `if`.

Improper comment after 'end if'

CompileIt! allows only a comment or end of line after an `end if`.

Incompatible reference parameter

When a ROM routine requires a "VAR" parameter, it must be a particular type. Either you are passing this ROM routine a Toolbox variable of a different type, a shared variable that you declared to be some other type, or else it is a local variable that you used in some incompatible way, perhaps by passing it to another ROM routine that requires a different VAR type.

Inconsistent or undefined value in 'it'

There is no value in `it`, or else it has been given different types in the two legs of an if-then-else.

Inconsistent use of variable

CompileIt! requires that the local variable used in a repeat with command be used only in arithmetic (integer) expressions. If you need to do string

operations on its value, copy it to another (dummy) variable for the purpose.

Inconsistent value in 'it' around repeat

The value in `it` is used at the beginning of the repeat, but it is not the same type at the end.

Internal error FT

Internal error R0

Internal error R4

Internal error R5

Internal error R6

Internal error Rxx

Internal error VP

All internal errors signify a consistency check that failed, either from a corrupted copy of *CompileIt!*, or from an undetected bug. If you continue to get this error after replacing *CompileIt!* from your backup, please contact Heizer Software.

Invalid function parameter list

Invalid command parameter list

Unlike interpreted HyperCard, *CompileIt!* requires that functions and commands within the ROM and the compiled script be passed exactly as many parameters as are defined for them. The ROM routines will bomb if passed the wrong number of parameters; the requirement in the compiled XCMD helps to make the machine code somewhat more efficient. This error message reports that there are probably too many parameters for a local command or function.

Invalid condition in if-then ('then' expected)

After the keyword `if`, *CompileIt!* looks for a Boolean expression, followed by the keyword `then`. If there is something wrong with the expression, *CompileIt!* may never reach the end of the expression before it starts looking for the `then`.

Invalid digit character

You will get this error if you try to do arithmetic on a `tab` or `return` or other character constant that is not a digit.

Invalid 'end' line

There is something other than an end-of-line or comment following your handler's name after the word `end`.

Invalid handler

Your handler does not begin with either `on` or `function`.

Invalid handler name

You have most likely used a reserved word as your handler name. Try renaming your function or handler.

Invalid parameter list

This indicates the wrong number of parameters were passed to a library routine. Maybe you've defined a custom symbol with the **Custom Symbol Edit** card and passed it the wrong number of parameters.

Invalid pass command

HyperTalk only allows you to "pass" the name of the handler in which it resides. This error occurs if *CompileIt!* finds a different name on the `pass` command.

Invalid repeat command

CompileIt! found something on the command line after it finished the recognizable parameters to the `repeat` command.

Invalid script structure

After the final end line of a script there should be nothing other than comments. There should also be nothing else between handlers when there is more than one handler in a script (except shared variable declarations).

Invalid variable name

CompileIt! gets very confused when you use the same name for a variable and the handler it is used in. It would help if you picked a different name for one of these.

Keyword 'of' expected after property name

This property or function is expecting some phrase telling what it is the property of, or what parameter to evaluate over. Maybe you meant something different?

Missing 'else' or 'end' for if-then

When the keyword `then` is the last word on its line, the sequence of commands that follow must be ended by either an `else` or an `end if`. *CompileIt!* found neither. Most likely there is something wrong with one of the commands.

Missing 'end' for if-then

When the keyword `else` is the last word on its line, the sequence of commands that follow must be ended by an `end if`. Most likely there is something wrong with one of the commands.

Missing end quote of string

A string constant must begin and end with a quote mark, must not contain a return or end of line character, and (in *CompileIt!*) must not be longer than 254 characters.

Missing Library routine <symbol>

CompileIt!'s internal library has become corrupted. Restore it from a backup copy. You can also get this error if you are running *CompileIt* in too little memory. Refer to the appendix about working in 1 megabyte for some possible solutions.

Missing or incorrect name at end of handler

The end line of a handler should have the same name as the first line.

Nothing to compile

This error occurs if there are no handlers in the script.

'of' expected in chunk expression

CompileIt! is looking for the syntax, *chunk of expression*. If you get the chunk part wrong, you could get this error.

One chunk at a time from handle or pointer

You cannot use a range of chunks (i to j) for *CompileIt!*'s handle and pointer chunk expressions, since one chunk is all you can get into a 68000 register.

Out of memory

CompileIt! has run out of memory. Refer to the appendix about working in 1 megabyte for some possible solutions.

Record field name expected after dot

Pure HyperTalk does not use a dot for anything except a decimal point. *CompileIt!* uses it to designate the subfields in a record data structure as defined in *Inside Macintosh*. But you must use the names listed in the defined records for that to work. Refer to the Name List screen in *CompileIt!* for the currently valid record subfield names (shown with an "R").

Record field name inappropriate here

Since the only record-sized data structures you can have in an XCMD created by *CompileIt!* will exist in the heap or shared variables, you will get this error if you try to treat local (or global) variables as if they were records.

Right parenthesis expected

It's possible that you have mismatched parentheses, but more likely that the expression inside the parentheses has a syntax error that makes *CompileIt!* assume it has reached the end prematurely.

Script is too big for one XCMD

The 68000 machine language has limitations that make it inconvenient to address code resources that are larger than 32K. This error reports that danger. If it happens, you should consider dividing your script into two or more smaller pieces.

Shared variable type expected here

The form for a shared variable declaration is:

```
global varname
```

or

```
global varname:sometype
```

It seems you put the colon in, but *CompileIt!* does not recognize a type word.

Simplify this expression

CompileIt! ran out of floating-point temporary variables, probably because integers or strings are being passed to local functions expecting floating point parameters. Pre-converting the parameter values (using ^1) before passing or breaking up a complex expression with put commands would help

Subscripts don't work here

There are several kinds of objects that cannot be given array subscripts (e.g., constants and local variables). You found one of these objects.

Subscript should end with ']'

Array subscripts are enclosed (on both sides) by square brackets. Perhaps you made a mistake in your index expression, and *CompileIt!* gave up too soon.

There is no object associated with an XCMD

The container `me` is not meaningful in the context of an XCMD.

There is no repeat for this next

CompileIt! found a next repeat command without an enclosing repeat command.

There is no repeat to exit

This error reports that *CompileIt!* found an exit repeat command without an enclosing repeat command.

There is no result here

The result is only meaningful in compiled code after a message is sent to another handler, or a function call, or after a div or mod. Some Toolbox procedures also return a result, but most do not. You can compile a small script with just the previous line (with whatever is required to support it) and *DebugIt* checked on, then step over the line and look at the result in the variable monitor to see if it has an indicated type. If the type is shown as “?” then there is no result to test.

‘to’ expected in repeat with

The repeat with command must have both a starting and ending value for the index variable. *CompileIt!* is looking for the keyword to (or the words down to) between these two values.

Too many parentheses

CompileIt!’s expression parser has a limit to how deeply you can nest parentheses and function calls — about 20. You probably exceeded it. Try breaking this line into separate lines with temporary variables.

Too many variables for one handler

CompileIt! has a limit to how many different local variables you can declare in one handler — about 200. It looks like you ran over. Try breaking this handler into separate routines called from a smaller dispatch handler.

Trap handler in ROM takes a different number of parameters

CompileIt! no longer permits you to compile a handler with the same name as a Toolbox routine, so you probably will not get this error message.

Use a number with handle or pointer chunk expression

CompileIt! can take handle and pointer chunk expressions when you tell it numerically which chunk you want, but it is not able just to take the last or middle (or any) chunk in this format.

Using 'the' for that does not work

The keyword `the` may only prefix a built-in function or a property. This error may turn up for legitimate functions or properties, too, if used improperly.

You can't pass to a ROM routine with VAR parameters

You should not try to compile an XCMD with the same name as a ROM routine, if that routine uses VAR Parameters. In general, you cannot compile externals with the same name as installed Toolbox names. If you absolutely need to have your external use the same name as a ROM routine, first compile it using a different name, then use ResEdit or Resource Mover to rename the external after compiling. But of course `pass` won't work, so to get the same effect you must just call the Toolbox routine directly, then exit (or return with the result).



APPENDIX E

COMPILEIT! VOCABULARY

This appendix documents *CompileIt!*'s vocabulary. The "Supported" words are symbols that *CompileIt!* understands when you first install it. *CompileIt!* can be "taught" about other symbols via the symbol table cards and the **Custom Edit** card. *CompileIt!* can also recognize commands and functions (including calls to XCMDs and XFCNs) and properties that are not part of its vocabulary even though it does not know what these symbols do — they will be converted into text callbacks.

CompileIt! is able to recognize when a name is a property based on how you use it. For this reason, only those properties which conflict with names in the ROM Toolbox are built into *CompileIt!* This is so an alert message will generally be displayed when a conflict occurs if you choose to install any of the Toolbox names.

Supported HyperTalk Vocabulary

Known Commands

add	drag	lock	select
answer	edit	mark	send
ask	else	multiply	set
beep	enable	next	show
choose	end	open	sort
click	exit	pass	start
close	export	play	stop
convert	find	pop	subtract
create	function	print	type
debug	get	push	unlock
delete	global	put	unmark
dial	go	putclickwait	visual
disable	hide	read	wait
divide	if	repeat	write
do	import	request	
domenu	inline	reset	

Known Functions and Properties

abs	itemDelim	msg	round
atan	itemDelimiter	number	second
average	left	numtochar	sin
bottom	length	offset	sqrt
bottomright	ln	param	tan
chartonum	ln1	paramcount	ticks
cos	log2	params	top
exp	max	random	toleft
exp1	menus	result	trunc
exp2	message	right	value
id	min		

Reserved Words

abbr	button	in	short
abbrev	card	into	stack
abbreviated	cd	is	the
after	contains	long	then
and	div	menu	there
background	field	menuitem	times
before	fld	mod	to
bg	for	not	until
bgnd	forever	of	while
bkgn	handle	on	window
btn	hypercard	or	with
			within

Valid Constants

colon	five	pi	tab
comma	formfeed	quote	ten
down	four	return	three
eight	linefeed	seven	true
empty	nine	six	two
false	one	space	up
			zero

Valid Chunks and Ordinals

all	fifth	line	seventh
any	first	lines	sixth
char	fourth	longint	tenth
character	integer	mid	third
characters	item	middle	wd
chars	items	ninth	word
eighth	last	pointer	words

CompileIt! also supports all of the standard operators (+, -, >, <, etc.). Unknown commands, functions, and properties that have normal syntax will be converted into text callbacks.

Unsupported HyperTalk Vocabulary

The following vocabulary words are not supported.

exit to HyperCard

Supported only under HyperCard 2.0 and above and only if the **HyperCard 2** option is turned on.

me

Me is not supported. (What would it refer to?) Use parameters to your external instead.



APPENDIX F

SUGGESTED READING

CompileIt! requires that you know how to write scripts in the HyperTalk language. If you're new to HyperTalk, reading one or more of the following books is suggested.

The Complete HyperCard Handbook, Danny Goodman, Bantam Computer Books, 1987

The Complete Book of HyperTalk 2, Dan Shafer, Addison-Wesley, 1990

HyperTalk 2.0: The Book, Dan Winkler and Scot Kamins, Bantam Computer Books, 1990

Steve Michel's SuperCard Handbook, Steve Michel, Osborne/McGraw-Hill, 1989

If you're looking for more information on XCMDs, including sample C and Pascal source code, you may want to read the following books.

XCMDs For HyperCard, Gary Bond, Management Information Source, Inc., 1988

HyperTalk 2.0: The Book, Dan Winkler and Scot Kamins, Bantam Computer Books, 1990

CompileIt! can be used to access the Macintosh ROM Toolbox. Using this feature requires an understanding of the Toolbox. It is strongly recommended that you have a working knowledge of *Inside Macintosh I-II*. Chernicoff's books also serve as excellent reference sources.

Inside Macintosh Volume I-VI, Apple Computer, Inc., Addison-Wesley, 1985

Macintosh Revealed, Part 1: Unlocking the Toolbox, Steven Chernicoff, Hayden/Apple Press, 1985

Macintosh Revealed, Part 2: Programming the Toolbox, Steven Chernicoff, Hayden/Apple Press, 1985

Macintosh Revealed, Part 3: Mastering the Toolbox, Steven Chernicoff, Hayden/Apple Press, 1989

How to Write Macintosh Software, Scott Knaster, Hayden/Apple Press, 1989

The Programmer's Apple Mac SourceBook, Thom Hogan, Microsoft Press, 1989

[REDACTED]

APPENDIX G

MEMORY REQUIREMENTS

The minimum memory required for *CompileIt!* 2.0 is 750K; ideally, you'll want 2000K.

CompileIt! has two modes of compiling — the **Slow Mode** and the **Fast Mode**. The **Slow Mode** is a little slower than the **Fast Mode** (about 30% with a good hard disk, much slower from other media) but can run very efficiently in a minimal amount of memory. The **Fast Mode** is faster but requires more memory. Each time you click the **Compile it** button on the **Script Card**, *CompileIt!* automatically decides which mode to use based on how much memory will be needed to compile the current script using the currently installed symbols. The **Compiler Phase Field** will let you know which mode is being used. You can also find out which mode will be used for the current script by going to the **Options Card**. (The field at the top left of the card will indicate whether the compiler will use the fast or slow mode of compiling.)

While all this is transparent, if you do not have lots of memory available, you can sometimes get *CompileIt!* to use the **Fast Mode** by shortening your script and/or removing some unneeded symbols from the symbol table. Removing unneeded symbols may also enable a script to compile if *CompileIt!* runs out of memory even in the **Slow mode**, which could happen with longer scripts. It may not be possible to compile moderately large scripts with *DebugIt!* turned on, or very large scripts at all, in a minimum memory configuration.

If you successfully compile a script with *DebugIt!* turned on, it will probably also run in the same memory space.

APPENDIX H

UPGRADING TO *COMPILEIT!* 2.x

This appendix is for *CompileIt!* 2.x users who have upgraded from an earlier 1.x version. New users do not need to read this appendix.

Here are a few tips to help make your transition to *CompileIt!* 2.x as painless as possible:

Preserving Custom Symbols

If you made use of the **Custom Symbol Edit** card in *CompileIt!* 1.5, then you will want to preserve the custom symbols you created and move them into *CompileIt!* 2.x. This is done via the **Save Names** button on the **Custom Symbol Edit** card. Just click on the **Save Names** button and an updater stack will be created for you with all of your custom symbols in it. This updater stack is compatible with *CompileIt!* 2.x. A button in the updater stack will add a new symbol card with all of your custom symbols on it into your copy of *CompileIt!* 2.x.

Batch Compiling

There are some subtle changes in the hooks for batch compiling; the former **Development Stack** has therefore been updated and renamed **Batch Compiling Stack**. Do not use the old **Development Stack** with *CompileIt!* 2.x.

New Syntax for Toolbox Access

CompileIt! 2.x offers some new syntax intended to simplify accessing the ROM Toolbox. Below is a script which takes advantage of the new syntax. This script is shown to illustrate how much simpler your code can be rather than to document new features, which is done elsewhere in this manual. It is recommended that you at least scan Chapter 5 *Getting to Know the Toolbox* to learn more about these features.


```

-- Display the color wheel
global RGBin:Record[6],RGBout:Record[6] -- shared variables

function SelectColor inRed,inGreen,inBlue
  setPt myPoint,0,0
  put inRed into RGBin.integertype[1] -- array notation
  put inGreen into RGBin.integertype[2]
  put inBlue into RGBin.integertype[3]
  if not GetColor(myPoint,"Please choose a color:",RGBin,RGBout)
  then return empty
  -- non-existent chunks are supported
  put BitAnd(RGBout.integerType[1],$FFFF) into item 1 of val
  put BitAnd(RGBout.integerType[2],$FFFF) into item 2 of val
  put BitAnd(RGBout.integerType[3],$FFFF) into item 3 of val
  return val
end SelectColor

```

APPENDIX I

WHERE TO FIND MORE INFORMATION ON *CompileIt!*

If you are like most *CompileIt!* users, you'll want to share tips and techniques, acquire more sample code, customize *CompileIt!* to suit your special needs, etc. This appendix will tell you where to find other users and how to get additional technical information on *CompileIt!*

Online Services

CompileIt! users can be found on Compuserve in the MacHyper forum ("Go MacHyper") and on America Online in the HyperCard section. Heizer Software will upload updater stacks, bug fixes, sample code, and technical notes to the software libraries at both of these online locations from time to time.

America Online has created a *CompileIt!* Special Interest Group where *CompileIt!* users can share tips and tricks, sample code, etc. Use Keyword "MHC" and then open the "Special Interest Groups" folder to find the SIG.

You can reach Heizer Software through several online services including America Online, Compuserve, and AppleLink. Check the membership directory of these services for Heizer Software's address.

User Groups

Heizer Software will be happy to provide *CompileIt!* technical notes and a Working Model edition of *CompileIt!* to your user group's library. The Working Model compiles only 10 lines or less, it does not include DebugIt!, and its symbol table cannot be modified. Call Heizer Software at 510-943-7667 for more information.

CompileIt! Linker — This utility creates "Code Libraries" which any *CompileIt!* user can use. Code Libraries are object code libraries similar in concept to MPW ".o" files. The Linker is required to create the Code Libraries, but once created, any user can install and use them. Simply refer to specific routines in a library and *CompileIt!* retrieves the object code for the named routine and incorporates it into your external. You can use many development systems besides *CompileIt!* to create your Code Libraries.

Use the Linker to distribute useful routines to other *CompileIt!* users while protecting your source code investment. Use with *CompileIt!* existing code you have written in other languages without translating your code into HyperTalk first. **#30-0427, \$49.00.**

About Updates

From time to time, Heizer Software may create updater stacks that fix minor bugs, add or update symbols (as new machines are introduced, additional ROM calls may be introduced as well), or add minor new features. You'll be able to find these updates in the MacHyper forum on Compuserve and in the HyperCard section on America Online.

Users who make use of the **Custom Symbol Edit** card in *CompileIt!* are encouraged to use the **Save Names** button on that card to create updater stacks that will be of use to other users. Hopefully, many users will create updater stacks and make them available through the major online services so that all users can benefit.

APPENDIX J

TIPS FOR CREATING USEFUL EXTERNALS

There are a number of things you can do to make your externals more useful to yourself and to others. Some of these tips are adapted from a document called "HyperCard's EDGE" available in the MacHyper forum on CompuServe.

Provide Help

It's not too hard to provide help to assist others in using your external. A common convention is to return help when the question mark ("?",) is passed to your external. It's also easy, as we shall see, to include such help in your *CompileIt!* externals.

Help is useful not only to others who might be using your external, but to you as well. It's easy, when you write your external, to know what parameters it takes, but it might not be so easy to remember them next month or next year. Since HyperTalk is always interpreted, it's easy to peek at the code to refresh your memory. You can't do that with compiled externals.

Suppose you've written an XFCN to sort lines in a container. You might include several optional parameters, so there's something for you to remember when you use the XFCN. Say your XFCN takes three arguments: the container to sort, the direction to sort it (i.e., ascending or descending), and the type of sort to make (i.e., an ASCII, International, Numeric, or Date sort). Here's how a syntax statement for this XFCN might read:

```
SortContainer (container, [direction], [type])
```

Here's how you would implement Help in this XFCN:

```
on Sort container,direction,type
  if container = "?" then return Help()
  --rest of code
end sort

function Help
  return "SortContainer(container, [direction], [type])"
end Help
```

Make sure your Help statement is concise and fits the standard of Apple's *Script Language Guide*. Also make sure it fits in one short line, so it can be read in the message box.

You can also include a second handler, to allow you to embed and return copyright information, when the exclamation point ("!") is passed to your external. This has the benefit of compiling, right into your external, the ASCII text of your copyright notices. This allows you at least to monitor use of your external in other stacks. Here's the function:

```
function copyright
    return "@1990 John Doakes. All Rights Reserved"
end copyRight
```

You can insert this line into your main handler, as was the Help test:

```
if container = "!" then return copyright()
```

Before using such a line, though, make sure you are providing the correct copyright information you want. "All Rights Reserved" has specific legal meanings.

Here are some other guidelines that should help make your externals easier to use:

Follow HyperTalk Syntax

Whenever possible, base the syntax of your external on the syntax of a related or similar HyperCard command or function. For example, HyperCard's `Offset` function takes two arguments, which are text strings, and returns the number of characters from the beginning of `string2` at which `string1` begins. If you are creating an XFCN that returns, say, the *line number* containing `string1`, you should make sure that the order of parameters passed works in the same way; don't reverse them.

Keep Input Short and Allow for Abbreviations

If your external wants some specific text, allow for abbreviations of that text. For example, don't require that the word "Help" be passed to your external. Allow instead for the question mark. In the earlier example about the sorting XFCN, don't require the word "Ascending", allow for an "A."

Allow Halting with Command-Period

HyperCard, of course, allows users to stop unconditionally any script with the Command-Period key combination, and your externals can do this too. This is especially true of externals that may take some time to run, but can come in handy at any time, such as during development. There is a function that tests to see if the Command-Period key combination is being pressed in the “Names for Toolbox Access” symbol card. You can call this function at any time in your external — preferably within a loop that might take some time to execute, and take appropriate action.

Avoid Object Dependency

Object dependency is referring to specific fields, buttons, cards, stacks, or other resources in your scripts. Instead of using lines such as `get the rect of card button 7` (which, as discussed in Chapter 3 of this manual, is a slow text callback), you can instead make that rectangle a parameter that is passed to your external. This makes for more general externals — that is, externals that can be used in many different situations — and makes for easier debugging and modification.

Don't Interact with the User

Generally speaking, your externals should not interact directly with the user, for instance by using the `Ask` and `Answer` commands to create a dialog. If your external needs information, make that information a parameter to the external; if your external is generating an error message, return that error message to the script.



APPENDIX K

INLINE MACHINE-LANGUAGE CODE

CompileIt! is intended to serve two kinds of users. Most users will not be expert programmers — in fact, we expect that with not much more sophistication than it takes to write good scripts in ordinary HyperTalk, you can use *CompileIt!* to write useful XCMDs. Most people using it at this level need never use the Toolbox ROM facility, and will still benefit from many of the advantages of compiled code.

A different group of users will know the ins and outs of the Macintosh, yet will find it simpler to use HyperTalk and *CompileIt!* than the conventional programming languages and their compilers. Part of the reason *CompileIt!* was written in HyperTalk was to ensure its ease of use.

With increasing sophistication also comes the occasional need to do something the compiler's author did not offer — perhaps to access some special resource in an unforeseen way. The best Macintosh compilers offer their users a facility called “inline code,” which lets the knowledgeable programmer specify actual machine language instructions in assembly language or hexadecimal constants. Suppose you want to call an FKEY from within your XCMD. *CompileIt!* does not know about FKEYs nor how to jump to them, and there are no Toolbox routines that do it. You need to use `INLINE`.

Apple's Pascal offers only hexadecimal constants for `INLINE` code. This is normally adequate, but you have to know a lot about what code the compiler is giving you and where the variables are to make it work. Because *CompileIt!* stores local variables offset from register A7 instead of A6 like most other compilers, variable access can be very tricky. The `INLINE` facility in *CompileIt!* lets you get the address of a variable or subroutine (handler or function) in a high-level notation that is less error-prone than straight hexadecimal. There is also notation for preserving the runtime stack integrity. This is important, since if you modify the stack pointer A7 without telling *CompileIt!* your XCMD will get wrong results or even bomb.

The form of the *CompileIt!* `INLINE` command is:

```
INLINE item,item,item,...
```


where each item is either a hexadecimal constant, a name reference, or a stack adjustment. Hex constants use the familiar "\$" notation, and should have no more than 4 digits (fewer is OK, they are filled out with zeros on the left). A name reference is the name of a local variable or handler compiled together in your script, preceded by a star "*"; the address of the variable or subroutine is loaded into register A0. The name must be already visible when you use it, that is, you have to already have put something into the variable, or the handler must precede the handler with the `INLINE`. A stack adjustment is a decimal number preceded by a "+" or "-", and should mirror your use of instructions that alter the value of A7. Thus if you code an instruction like `$598F` that decrements the stack pointer by 4 (effectively increasing the stack size by 4 bytes), you should follow it immediately by the adjustment `+4`.

For example, many people use the FKEY "Switch-A-Roo" on their Mac II to switch to 2-color mode so that the visual effects will work. To call up this FKEY, it is necessary to load it into memory with a `GetResource` ROM call, lock it down with `HLock`, and jump to it with a `JSR`. The `JSR` part requires `INLINE`. The script will look something like this:

```
on FKEY num
  put GetResource("FKEY",num) into myHandle
  HLock myHandle
  INLINE @myHandle -- handle to A0;
  INLINE $2050     -- MOV @A0,A0
  INLINE $2050
  INLINE $4E90     -- JSR @A0
  HUnlock myHandle
  ReleaseResource myHandle
end FKEY
```

APPENDIX L

INSIDE *COMPILEIT!*

This appendix peeks under the covers of *CompileIt!* for people who need the extra technical information or are just plain curious. There are two kinds of “inside information” of interest, which we address in two sections. The first is concerned with what kind of code *CompileIt!* generates for your external. This is of interest to machine language techies trying to interface their XCMDs to other languages, or those wanting to use *CompileIt!* to compile code resources other than HyperCard externals. Heizer Software does not support these unconventional uses of *CompileIt!*, but they are possible if you know what you are doing. This appendix tries to help. The second section gives some information about how *CompileIt!* itself works — mostly to satisfy curiosity, since there are no user-serviceable parts inside.

Inside the *CompileIt!*-Compiled External

The compiled external consists of four sections: (1) the startup prolog, (2) the string constants, (3) the compiled handlers and functions, and (4) the included library. All versions of *CompileIt!* generate code resources in the same form.

The startup prolog is mostly canned machine code with a JSR instruction to the main function or handler inserted at compile time. The startup code saves all the conventionally non-volatile CPU registers (HyperCard saves them also before jumping to the external, but other hosts may not), and initializes its own register defaults. The string reference count handle is allocated and initialized, and if shared variables are used, they are cleared to zeros (by a library call). The call to the main handler is carefully placed so that it can be recognized by the inline XCMD linker at runtime, if you use the direct XCMD call feature of the Custom Symbol Edit Card. The linker code depends on the fact that the prolog has a special form, and we have carefully preserved that form in *CompileIt!* 2.0. When the main handler exits, it returns to the prolog, which then disposes the string handles no longer in use, restores the saved registers, and returns to HyperCard (or other host).

The prolog code is constructed from two templates in the first four lines of the LIBR #16383 resource. One template is for the pure HyperTalk external, and allocates less space on the CPU stack. The other is used when the script has Toolbox calls or shared variables, and accomodates the increased memory

requirements of these features. If you use MacsBug or another debugger that knows about MacsBug names, the prolog (except for the first three instructions) will be identified as “XCMDglue.”

The string constant section needs little explanation: it consists primarily of P-strings, with occasional SANE constants. If *CompileIt!* discovers a constant during Pass 2 that was not known in Pass 1, the constant is simply coded inline, with a branch over it.

The compiled code section consists of one or more closed subroutines. Each routine begins with a LINK instruction whose sole purpose is to aid MacsBug to find its name; it is not executed. Following the LINK is a sequence of CLR instructions that allocates local variable space on the stack and simultaneously clears it to zeros. If the routine uses any SANE operations, some additional (uninitialized) space is allocated for expression temporaries. After that comes the compiled code itself. Each subroutine ends by popping off the stack any temporary variable space, then loading into D0-D2 any result before exiting with an RTS. When MacsBug names are enabled, the RTS is followed by an otherwise unused UNLINK and another RTS to flag the handler name.

The library code similarly consists of a sequence of closed subroutines, all selected from a canned set of code fragments in the LIBR #16383 resource. At the beginning and end is the requisite (but otherwise unused) LINK and UNLINK instructions to give the entire sequence the name “Library.”, though the late addition of the itemDelim property in this space (just before the “L”) tends to foil MacsBug. The order of the library routines is determined by their references in the library section of the current symbol table (LIBR resource #16384), but the selection is dynamic — only library routines that are actually used are included. The current item delimiter, as set by the ItemDelim property, is stored in the first character of the library name, just before the “L.” The library is all hand-coded machine code. We expect future versions of *CompileIt!* to be good enough to permit the library routines to be written in HyperTalk also, eliminating this one tiny fib in the source code.

Compiled HyperTalk Code

The best insight into the kinds of code generated for a particular HyperTalk command or operator is to examine the code information line in the DebugIt! window.

Calls to local handlers and functions are the simplest code. The parameters are

pushed onto the stack as 4-byte values or handles, and the function or handler result is returned in the D0 register. The caller is responsible for removing the pushed parameters. String handles passed as parameters are reference-counted before the procedure is called, and counted back down by the called procedure before it returns. Toolbox calls are generally very similar, but the size of the parameters depends on the data types, and the called toolbox routine is responsible for removing the stacked parameters. The symbol table entry defines how many bytes are required for the result and designates the stack as where the result is when it comes back.

OS Toolbox calls are register-based. In the general case, the parameters are pushed onto the stack as before, then popped into the requisite registers as specified in the symbol table glue code. When an OS routine takes only one parameter in a register, the symbol table has a way to designate that register as the destination register directly, eliminating the extra stack push and pop. The specific symbol table notation for this is discussed in technical notes elsewhere.

The interface to callbacks is also fairly simple. The parameters to callbacks are stored directly into the XcmdBlock, and return value is extracted from the OutArgs. Because the parameters are not pushed on the stack, nobody is responsible for removing them.

In some cases additional glue is required to convert between the public and private interface to a Toolbox routine or callback. For example, each of the packages and many of the newer managers have a single Toolbox trap, and the many routines are selected by a 2-byte selector pushed on the stack or loaded into D0. Since a different selector is used for each entry name in the group or package, it is encoded as part of the glue in the symbol table. Another complication arises from certain kinds of VAR parameters, and this is also solved in the glue code. These are not really compiler code generator problems in the sense that the compiler is not concerned with them, and they are solved elsewhere.

The rest of the code generation is strictly the compiler's domain. In the case of data type conversions, the compiler has a roadmap that connects every data type it knows about to every other data type. Some times the road from type A to type B goes through type C, and sometimes there is a direct route. Most data type conversions involve extensive code, so a toolbox, callback, or library routine is used; again, the DebugIt! code information line is probably the best way to discover what is happening, or for a more detailed analysis you can disassemble sample code lines after they are compiled.

When you test a comparison in an IF command, *CompileIt!* generates the correct conditional branch for the condition being tested. If the condition is used in a logical operator such as AND, OR, or NOT, or if its result is put into a variable (including passing it as a parameter to a function or handler), the condition is converted first to a single bit in a 32-bit word using the SETcc opcode and a couple of additional instructions to extend the byte to a full word. Almost all forward branches generated by *CompileIt!* are the longer form.

Repeat loops use the same condition testing mechanism as the IF command, when that is appropriate (`repeat while` or `repeat until`). If the repeat is given a fixed number of repeats, by either the WITH or TIMES form, the number of repeats is pushed onto the stack, and counted by a SUBQ instruction. In repeat with, the control variable is independently incremented or decremented, so that if the compiled script should modify the control variable within the loop, this does not affect the actual number of repeats — just like HyperCard. DebugIt! displays the contents of the stacked repeat counters for active repeats in the current handler script at the top of its variable window.

Integer arithmetic is one of the few places where *CompileIt!* can generate efficient native code. Multiplying or dividing by a constant power of 2 is recognized by *CompileIt!*, which generates arithmetic shift instructions. Dividing a negative number in HyperCard gives a slightly different result than a simple shift, so additional glue is required to correct the result in those cases. If you understand what you are doing, or all your divides by a constant power of 2 involve positive dividends, then you can set a special flag that eliminates the extra glue. This flag also predisposes the result of a local “on” handler to be integer (not string), so be careful if you are using the result of calling such handlers. The flag may be turned on by replacing the “Prelude(8)” in the LoadnGo call in the first handler of the stack script with “Prelude(24).” The meaning of these flags is defined fully elsewhere.

Division by a variable or expression other than a constant power of 2 is performed by a library routine. Division by zero will typically give some wrong answer (any answer is of course wrong), but the result will contain the string “DIV 0” instead of the normal empty string. Note: do not attempt to modify the result from a divide in DebugIt!, or you will trash library code.

The 68000 CPU has only a 16x16-bit multiply operator, so *CompileIt!* generates 3 multiplications and adds the results to get a correct 32x32-bit multiply with a 32-bit result. A fourth multiplication would have been required to

obtain the complete 64-bit result, but since *CompileIt!* discards the high 32 bits, this is not necessary. The multiplication code is compiled in-line.

If you use the Add or Subtract command with a local integer variable and a small integer constant (as in “add 3 to x”), *CompileIt!* will generate optimal 68000 code, a single ADDQ instruction. Otherwise all integer arithmetic is performed in the data registers of the CPU. Better code is sometimes possible, but the cost in compile time is not deemed worth it at this time.

Local variables are accessed in the compiled code by an offset from A7. Since temporary values may be pushed onto the stack (which also uses A7) from time to time, *CompileIt!* maintains a current stack depth that is used to determine the correct offset to local variables, and also to determine how much to discard when the handler or function exits. This is why it is important, when you use INLINE commands, to accurately report to *CompileIt!* the migration of A7 caused by your code. Shared variables are pushed onto the stack in the XCMD prolog, and are accessed by offsets from A6, which is not altered by the individual handlers.

CompileIt!'s Toolbox extensions to HyperTalk gain the advantage that these constructs bring to C and Pascal: very efficient code. Access to shared variables and Toolbox globals is as fast and efficient as local variables. When you use “@”, “.”, or the array notation “[...]”, the code generated is nearly as good as your favorite MPW compiler...and getting better as we work on it.

Using MacsBug, you can disassemble by doing the following: Compile the script with DebugIt! enabled, and insert a Debugger or DebugStr command just before the line of interest. Run the XCMD, and when you stop in MacsBug, you can disassemble from the current PC to see the next line of object code. By doing this with DebugIt! turned on, it is easy to see where each line starts and ends. Between the lines *CompileIt!* inserts DebugIt! calls, each consisting of a MOVQ line#,D0, followed by a library call. You can count the number of lines from the start of the script to the line of interest to identify which MOVQ is the line number for the line of interest.

Inside the Compiler

CompileIt! uses a textbook recursive descent (top-down) parser to analyse the commands and expressions of the HyperTalk being compiled. Because the first implementation had to run in Apple's HyperCard environment with its “Too much recursion” limitation, the recursion stack of the parser was trans-

lated into a data structure, originally a card field, then a faster string variable, but now a simple handle on the heap. Other heap-based structures track nested loop structure, and in the case that DebugIt! is enabled, the line statistics for incorporating into the listing part of the DebugIt! window. Handles also accumulate constant strings during Pass 1 so that they can be inserted into the output binary resource in a single location, the locations of the EXIT and RETURN commands, a separate expression stack for SANE expressions, a compiler stack used to remember where temporary expression values exist, and the like. Handles are also used for local copies of the current script, symbol table, and output code structures.

For each function or handler being compiled, *CompileIt!* builds a symbol table line (in memory only, not added to the resource) in the same format as the symbol table Toolbox functions and procedures. A HyperCard string handle is constructed with the names of all the parameters and local variables. Thus the compiler limit of these variables is quite generous. Another handle is constructed to record the places where this procedure is called; a similar list is constructed for each library routine.

The symbol table is searched in two steps for any symbol encountered in the script. The first phase searches through the symbols defined in the current script in a linear order, generally last symbol first. If not there, the main symbol table is scanned by a fast binary search, and if the symbol is found, its location is added to a hash table for even faster recovery next time. This limits the effect of the number of installed symbols on compile speed from searching, though local names added to the front still require shifting the whole table as the symbols are added.

CompileIt! allocates another 2K of fixed-size P-string and other temporary variables, including the current token (word, constant, or punctuation), the current symbol table line, the name of the external and of the current handler, and an expression type stack. These are allocated on the CPU stack in the place of shared variables, as are also most of the compiler's internal globals.

When *CompileIt!* starts up, a small XCMD (LoadnGo) does much of the initialization. If the available memory is sufficient (as determined by bit 1 in the option parameter being 0), LoadnGo builds an integrated code block from KODE resources. Otherwise it loads the kernel for the paged version. In the integrated version all calls are resolved within the same KODE resource, while in the paged version the most common routines are in the kernel and the others cause the currently active overlay to be swapped out and replaced by the KODE resource containing the called routine.

CompileIt! does not support swapping in your externals, but the paged mode of compiling used for tight memory situations actually swaps segments of code in and out as *CompileIt!* works. It may be of interest that the segment manager within *CompileIt!* converts the return addresses on the stack into relative addresses for inter-segment calls, then restores the absolute addresses for the reloaded segment in a possibly different location on return, perhaps a unique implementation in Macintosh software. The result is that *CompileIt!* can operate (in the paged mode) with less than 50% of the memory required for the resident (integrated mode) compiler. Careful selection of the routines for each segment enables paged execution to run a mere 30% to 50% slower than the integrated mode. The extra processing time is mostly due to disk access, and a faster hard drive will suffer less degradation than (for example) a floppy disk.



APPENDIX M

THE HYPERCARD 2.X

XCMD INTERFACE

The definitive source for information on HyperCard external commands and functions is the *Claris Script Language Guide* (SLG). This appendix, however, summarizes the most important and useful facts. XCMDs and XFCNs are code resources that **CompileIt!** installs in the resource fork of the target stack. They lie in the standard HyperCard message hierarchy between the stack script and the first of any “StacksInUse” (or the Home stack, if none). Thus a handler in a card, background, or stack script will have the first chance at a command or function message, then the installed XCMDs and XFCNs, then the uncaught message continues up the hierarchy to the stacks in use, home, and finally to HyperCard itself. The external can also “pass” a message it caught on up the hierarchy, just as any other script can.

When HyperCard finds an XCMD or XFCN with a resource name that matches a message not caught by the stack script or earlier, it loads the resource into memory, constructs a data structure called the XcmdBlock, and sends that to the code in the resource. All this is transparent to both the scriptor and the writer of the external using **CompileIt!**, since the form of message handlers and functions in the external is so similar to HyperTalk in the stack scripts. If you need to use the fields of the XcmdBlock, they are defined in the **HyperCard 2 Names** card of the symbol table options, and listed at the end of this appendix.

HyperCard provides the external with a collection of useful functions and procedures that the external can invoke, called *callbacks*. Some callbacks represent operations that only HyperCard can do, because they access private data structures. Others are simply utilities that HyperCard must support for its own use, and are made available to the rest of us as a convenience. **CompileIt!** automatically calls some of the callbacks as a part of the generated code for your compiled scripts; these are noted in the symbol card. Other callbacks are duplications of code that we think **CompileIt!** can do better in its own library. Still others are necessary to support external windows in HyperCard 2.0 and later.

All the callbacks we know about are listed in the next section, with a brief explanation of how to use each one. The Claris documentation shows the

callbacks as Pascal functions and procedures, but we have listed them using HyperTalk syntax, since that is how you would use them in *CompileIt!* There may be additional callbacks included in the **HyperCard 2 Names** card but not listed here. Two undocumented (and probably unsupported) callbacks are described in Winkler and Kamins, *HyperTalk 2.0: The Book*.

The current edition of the *SLG* no longer publishes the request numbers for the callbacks, but they are easy enough to extract from Apple's ".o" file or the *CompileIt!* symbol table. Users of *CompileIt!* probably don't need these numbers anyway, except possibly to decode the cryptic numbers in DebugIt!'s "Review Binary Callbacks" dialog. The callbacks *CompileIt!* puts in the external are decoded in the dialog for you, and the others can usually be inferred from the name of the callback in the currently executing line of your source code.

HyperCard 2.0 introduced a new interface for externals that allows them to receive messages designated for a particular window that the external might own. These messages give the external a flexibility not unlike the objects on a card, which the scriptor can program to respond to mouse clicks and other events. There is a defined set of messages that HyperCard can send to an external which is controlling a window. These are listed, with a brief explanation of their significance, in the section following the callbacks. Again, we have shown the HyperTalk syntax *CompileIt!* uses to support these events, rather than the Pascal code suggested by the HyperCard documents.

Events sent to external windows come with parameters, somewhat like the message that any XCMD or XFCN would receive, and we have tried to make the interface as much like HyperTalk as possible. In some cases, however, the data structure does not lend itself to a simple representation as a parameter list. The final section of this appendix, therefore, gives the component fields of each interface structure, including the XWEventInfoPtr block, with a brief explanation of its fields. The fields are shown using type indications as if they were shared variables; they are only fields in a record structure, but the notation may help us understand their types and sizes.

All of the callbacks, constants, and data structures listed in the *SLG* for the use of externals are included in the **HyperCard 2 Names** card of the symbol table options. The names that you are unlikely to need are marked (•) so that you can install the entire list, then **Remove Marked** to eliminate the useless ones.

The *SLG* includes the source code for the XCMD "Flash" in Pascal, C, and

Assembly. The **Examples** card of *CompileIt!* gives the source code for an equivalent XCMD, but in HyperTalk for comparison purposes.

HyperCard Callbacks

on AbortScript

This callback works something like executing the command `exit to Hypercard` in a script, except that it only takes effect when your external returns to HyperCard. HyperCard assumes it is not safe to abort your machine code, and besides, if you asked for this, then presumably you can figure out how to get back to HyperCard safely. This callback is intended for use in a debugger xWindowid, but *CompileIt!* also uses it for the `exit to Hypercard` command.

on BeginXSound windowPtr

Use this callback before making any direct sound manager calls. If calling from an external window, pass your `WindowPtr` parameter so that HyperCard can send you a `GiveUpSoundEvent` if necessary. If you pass `nil` (perhaps because your external has no window), then HyperCard will not be able to recover the sound channel if it needs it.

on BeginXWedit windowPtr

This callback lets HyperCard know that your window will be the active editing environment, so you will thereafter get all edit and keystroke events (except for commandKey menu equivalents).

on BoolToStr bool, str

Convert the Boolean value `bool` to a P-string in the variable `str`.

on CloseXWindow windowPtr

This callback lets HyperCard know that you wish to close your window. You should not actually close anything yourself until your external receives a `CloseEvent`, and then you should only dispose your own structures, but let HyperCard close and dispose of the window itself. HyperCard owns the `windowPtr`, it's only on loan to you. You could also call `CloseXWindow` for some other XCMD's window if you have the chutzpah; it's the same as typing `close window whatever` in the message box or a script.

on CountHandlers handlerCount

This callback returns in the integer variable `handlerCount` the number of active handlers. Zero means that no script is currently running.

on EndXSound

Use this callback after completing all direct sound manager calls, including deallocating the sound channel.

on EndXWEdit windowPtr

This callback lets HyperCard know that you are ready to give up editing. It will respond by sending your window a `GiveUpEdit` event, which can also happen if another window requests `BeginXWEdit`.

on ExtToStr num, str

Convert the Extended (SANE) value `num` to a P-string in the variable `str`.

function EvalExpr(expr) -- HyperCard string result

Evaluate the str255 (P-string) expression `expr`, and return a zero-terminated HyperCard string handle result, which *CompileIt!* recognizes as an ordinary string.

on FormatScript scriptHndl, insertionPt, quickFormat

This uses HyperCard's script formatting routine to reformat the script in the zero-terminated handle `scriptHndl`. If `quickFormat` is true then only the handler containing the insertion point is formatted. You should pass the byte offset to the insertion point in the variable `insertionPt`, which HyperCard will adjust to a new value reflecting the same relative position in the text. If you are using `TextEdit`, you can pass it `TEhandle@.selStart`, then call `TESetSelection` after `FormatScript` returns. If you have the script as a HyperCard string, you can get its handle by using `CharsHandle`.

function FrontDocWindow -- windowPtr result

This returns the `windowPtr` of the front window in the document (not floating) layer.

```
function GetCheckPoints -- handle result
```

This function returns a handle with a copy of the checkpoints for the window's script, or nil if there are none. The handle is an array of 16 integerTypes. You must dispose the handle yourself when you are finished with it.

```
function GetFieldByID(cardFld,fldID) -- HyperCard string result
```

If cardFld is true, then return the contents of the card field whose ID is fldID as a HyperCard zero-terminated string; if false, then return the contents of the background field instead.

```
function GetFieldByName(cardFld,fldName) -- HyperCard string
```

If cardFld is true, then return the contents of the card field whose name is fldName as a HyperCard zero-terminated string; if false, then return the contents of the background field instead.

```
function GetFieldByNum(cardFld,fldNum) -- HyperCard string  
result
```

If cardFld is true, then return the contents of the card field number fldNum as a HyperCard zero-terminated string; if false, then return the contents of the background field instead.

```
function GetFieldTE(cardFld,fldID,fldNum,fldName) -- TEhandle result
```

If cardFld is true, then return a copy of the styled TEhandle for the card field whose ID is fldID, or if fldID is zero, then return the TEhandle for card field number fldNum, or if that is zero also, then return it for the card field whose name is fldName. If cardFld is false, then return the styled TEhandle for the corresponding background field instead. If the field does not exist, or there is insufficient memory to allocate a new TEhandle, nil will be returned. Note that this is a copy of the master TEhandle, so you must call TEDispose when you are done with it. Note also that its grafPort (theHandle@.inPort) is left pointing to the current card, so if you do any drawing with this handle (including sending it to TEUpdate or TEsSetSelect), you may want to change that field to prevent drawing on the card window.

```
function GetFilePath(fileName,numTypes,typeList,askUser,~  
fileType,fullName) -- Boolean result
```

This uses the HyperCard search paths as specified in the Home stack to

locate the full path name for the file `fileName`. The `numTypes` and `typeList` parameters are as in the `SFGetFile`. If `askUser` is true, then if HyperCard cannot find the file on its own, it will put up a dialog to ask the user. The full path name of the found file is returned in the string variable `fullName`, and its type is returned in the variable `fileType`. If the file is not found, then `GetFilePath` returns false.

```
function GetGlobal(globName) -- HyperCard string result
```

Returns the value of the HyperCard global variable `globName`.

```
on GetHandlerInfo handlerNum, handlerName, objectName, varCount
```

This callback returns information about one of the currently active handlers. If `handlerNum` is 1 then the information is the last one called that has not yet returned; 2 is the script that called it, and so on back to the script that received the original mouseUp or other system message. The name of the designated handler is returned in the variable `handlerName`, and the variable `objectName` gets the name of the object it belongs to. The integer variable `varCount` gives the number of variables that handler knows about. If there are no active scripts, then calling `GetHandlerInfo` with 0 in `handlerNum` returns the number of defined global variables in `varCount`.

```
function
```

```
GetNewXWindow(templateType, templateID, colorWind, floating)
```

This opens an external window ("xWindoid") from the resource number `templateID` of type `templateType` (which must be either `WIND` or `DLOG`), and returns the `windowPtr` as the function result. If `colorWind` is true and Color QuickDraw is not present, then no window is opened and `GetNewXWindow` returns nil. The window is opened in HyperCard's floating or document layer, depending on whether `floating` is true or false. The window pointer is used to identify which window a subsequent window event belongs to.

```
on GetObjectName XTalkObjectPtr, objName
```

This command returns in the string variable `objName` the name of the object that has its data defined by `XTalkObjectPtr`.

```
on GetObjectScript XTalkObjectPtr, scriptHndl
```

This command returns in the handle variable `scriptHndl` the script of the object that has its data defined by `XTalkObjectPtr`.

```
function GetStackCrawl -- HyperCard string result
```

This returns a string containing the list of open handlers, indented as in the message watcher.

```
on GetVarInfo handlerNum, varNum, varName, isGlobal, varValue
```

This callback returns information about one of the variables known by the specified handler. The parameter `varNum` should be a number between 1 and the `varCount` returned by `GetHandlerInfo` for the same `handlerNum`. The name of the designated variable is returned in the variable `varName`, and the Boolean variable `isGlobal` is set to true if the variable is global. The HyperCard string variable `varValue` gets the value of the variable.

```
on GetXResInfo resFile, resID, rType, name
```

HyperCard returns in variable `resFile` the file reference number of the resource fork that the external was read from, and the resource ID, type (XCMD or XFCN), and name in the other three variables.

```
on GoScript
```

This callback is intended to enable a debugger to tell HyperCard to resume normal execution.

```
function HCWordBreakProc() -- procPtr result
```

This function returns the pointer to HyperCard's internal word-break procedure, which it uses for editing fields, scripts, etc. You can use this in the `wordBreak` field of a `TextEdit` record if you wish.

```
on HideHCPalettes
```

This callback tells HyperCard to hide all its built-in palettes.

```
on LongToStr posnum, str
```

Convert the unsigned `LongInt` value `posnum` to a P-string in the variable `str`. Ordinarily *CompileIt!* considers all integers to be 32-bit signed numbers, but dates after 1971 (calculated in seconds) come out negative. You can use this callback to return positive strings for dates more recent than that.


```
function NewXWindow(boundsRect,title,visible,procID,~
    colorWind,floating) -- windowPtr result
```

This opens an external window ("xWindoid") using the specified parameters, and returns its windowPtr as the function result. The parameters `boundsRect`, `title`, `visible`, and `procID` are as in the Toolbox call, `NewWindow`. For example, if `visible` is false, then the window is created but not shown. If `colorWind` is true and Color QuickDraw is not present, then no window is opened and `GetNewXWindow` returns nil. The window is opened in HyperCard's floating or document layer, depending on whether `floating` is true or false. The window pointer is used to identify which window a subsequent window event belongs to. To create palettes like HyperCard's, you can use the following constants for `procID`:

```
2048    paletteProc -- window with grow box
2052    palNoGrowProc -- standard window
2056    palZoomProc -- window with zoom and grow boxes
2060    palZoomNoGrow -- window with zoom but no grow box
8       hasZoom
2       hasTallTBar
1       toggleHilite
```

on Notify

This callback causes HyperCard to blink a small stack icon over the Apple menu if it is running in the background. It does not return until the user switches HyperCard to the front.

on NumToHex num,nDigits,str

Convert the integer value `num` to a hexadecimal P-string of length `nDigits` in the variable `str`.

on NumToStr num,str

Convert the integer value `num` to a P-string in the variable `str`.

```
function PasToZero(str) -- HyperCard string result
```

Convert the P-string parameter `str` to a native HyperCard null-terminated string.

on PointToStr pt, str

Convert the Toolbox point in `pt` to a P-string in the variable `str`.

on PrintTEHandle TEhandle, headerStr

This displays a print job dialog, then prints the TE record with the font, size, and style information in it, with `headerStr` as a header string.

on RectToStr rct, str

Convert the Toolbox rectangle in `rct` to a P-string in the variable `str`.

on RegisterXWMenu windowPtr, menuHandle, registering

Use this callback to associate a menu with a particular window. After you create a menu with a unique `menuID`, call `RegisterXWMenu` with your `windowPtr` and the `menuHandle`, and `registering` set to true. Then until you call `RegisterXWMenu` again with `registering` set to false, your window will receive a `MenuEvent` each time the user selects an item from that menu.

on ReturnEventResult theResult

If you use *CompileIt!*'s built-in event management, you have no need for this routine. However, if you wanted to roll your own, this handler takes a HyperCard string, detaches it from *CompileIt!*, and stores it into the `EventResult` field of the `eventInfoPtr`.

on ReturnToPas zeroStr, str

Copies characters from memory starting at the address in `zeroStr` to the P-string variable `str`, stopping at the first return or null, or when 255 characters have been copied. Note that `zeroStr` is a pointer, not a handle or string.

on RunHandler handler

If `handler` contains a regular message handler beginning with `on` then temporarily insert that handler in front of the current card in the message hierarchy and run it, otherwise execute whatever sequence of commands is in the string as if they were typed into the message box.

on SaveXWScript scriptText

This stores the HyperCard string `scriptText` back in the stack as the script for the object attached to this script editing window. If you have a handle instead of a string, you can use `ZeroTermHandle` or `HyperCardText` to convert it. Scripts are limited to 32K.

on ScanToReturn scanPtr

Starting at the address in variable `scanPtr`, look for a null or return character, and set `scanPtr` to the address of the null or just past the return.

on ScanToZero scanPtr

Starting at the address in variable `scanPtr`, look for a null and set `scanPtr` to its address.

on SendCardMessage themsg

Send the P-string `themsg` to the current card.

on SendHCEvent Eventrecord

If you are calling `GetNextEvent` or `WaitNextEvent` yourself, you must use `SendHCEvent` to pass to HyperCard any update events that are not for your window, and all app4evts.

on SendHCMessage msg

Send the P-string `msg` directly to HyperCard.

on SendWindowMessage windPtr,windowName,themsg

This is the same as writing `send themsg to window windowName in HyperTalk`, except that if you give it a window pointer in `windPtr` instead of nil, the name is ignored.

on SetCheckPoints checkLines

A script editor uses this callback to set the checkpoints on the script in its window. The handle `checkLines` is a 32-byte structure with 16 integerTypes, one for each checkpoint.

on SetFieldByID cardFld, fldID, fldVal

If `cardFld` is true, then replace the contents of the card field with ID `fldID` by the HyperCard string `fldVal`; if false then replace the contents of the corresponding background field instead.

on SetFieldByName cardFld, fldName, fldVal

If `cardFld` is true, then replace the contents of the card field with name `fldName` by the HyperCard string `fldVal`; if false then replace the contents of the corresponding background field instead.

on SetFieldByNum cardFld, fldNum, fldVal

If `cardFld` is true, then replace the contents of card field number `fldNum` by the HyperCard string `fldVal`; if false then replace the contents of the corresponding background field instead.

on SetFieldTE cardFld, fldID, fldNum, fldName, fieldTE

Set the text and styles of the designated card or background field to match the text and styles in the TEhandle `fieldTE` (see `GetFieldTE` for rules for deciding which field). You must dispose the `fieldTE` handle yourself.

on SetGlobal globName, globValue

Set the value of the HyperCard global variable `globName` to the string `globValue`.

on SetObjectScript XTalkObjectPtr, scriptHndl

This command replaces the script of the object that has its data defined by `XTalkObjectPtr` with the script in the zero-terminated handle `scriptHndl`.

on SetVarValue handlerNum, varNum, varValue

This callback replaces the contents of the specified variable with the string in `varValue`.

on SetXWIdleTime windowPtr, ticks

If your external window needs periodic execution time, you can call `SetXWIdleTime` and specify how often in the `ticks` parameter. Thereafter you will get `nullEvents` when HyperCard is idle (even in the background), until you call it again with zero ticks.

```
function ShowHCAalert(dlgID,promptStr) -- integer result
```

This function calls up one of the four standard HyperCard alert dialogs, identified by the number 1, 2, 3, or 4 in `dlgID`. The function result is which button the user clicked. If you use the ToolBox call `ParamText`, you can also use the wildcard strings “^0”, “^1”, “^2”, “^3” in your prompt string. Here are the choices (button 1 is always default):

dlgID	dlgID name	constantButtons
1	errorDlgId	1: OK
2	confirmDlgId	1: OK, 2: Cancel
3	confirmDelDlgId	1: Cancel, 2: Delete
4	yesNoCancelDlgId	1: Yes, 2: Cancel, 3: No

```
on ShowHCPalettes
```

This tells HyperCard to show the palettes that were hidden by a previous call to `HideHCPalettes`.

```
function StackNameToNum(stackName) -- integer result
```

This returns a unique internal reference number for the named stack. The reference numbers are only valid until HyperCard quits.

```
on StepScript stepInto
```

This callback enables a debugger to tell HyperCard to execute one line of the current script. If `stepInto` is true, then if this line of HyperTalk sends a message that is caught by another handler, it causes that handler's script to open.

```
function StringEqual(str1,str2) -- Boolean result
```

Compare the two P-strings, and return true if they are equal.

```
function StringLength(strPtr) -- integer result
```

Count the number of characters from the address in `strPtr` to the next null. Note that `strPtr` is a pointer, not a handle or string.

function StringMatch(pattern,target) -- pointer result

Search the characters beginning at the address in `target` for a string matching `pattern`, and return its address if found, or else return zero if a null is found first.

function StrToBool(str) -- Boolean result

Return the P-string value `str` as a Boolean.

function StrToExt(str) -- SANE result

Return the P-string value `str` as a SANE extended.

function StrToLong(str) -- integer result

Return the P-string value `str` as an unsigned long integer. Note that *CompileIt!* cannot distinguish unsigned numbers from signed numbers. Add, subtract, and multiply work the same for both, but compare and integer divide will usually give different answers for numbers larger than 2,147,483,647 (because they look like negative numbers).

function StrToNum(str) -- integer result

Return the P-string value `str` as a (signed, long) integer.

on StrToPoint str,pt

Convert the P-string `str` to a Toolbox point and return it in the variable `pt`.

on StrToRect str,rct

Convert the P-string `str` to a Toolbox rectangle and return it in the variable `rct`. Note that *CompileIt!* does not allow local variables of type Rect, so you must either use a shared variable of type Rect, or else allocate space on the heap and pass a dereferenced pointer or handle. You can also pass this callback the reference to a field of type Rect in a data structure such as a `GrafPort`.

on TraceScript traceInto

This callback enables a debugger to tell HyperCard to continue executing the current script, one line at a time.

on XWAllowReEntrancy windowPtr,allowSysEvts,allowHCEvts

The externals produced by *CompileIt!* have always been completely re-entrant, that is, it is safe for a text callback to result in a second event (or another normal call) being sent to your external before you return from the first. Other language compilers (most notably C) store information in their own code resource in an unsafe way, so they are accommodated by HyperCard (which has no idea which compiler was used). DebugIt! also stores some important information in its resource, but it makes a note of that in the global variable HyperDebugIt, so you need not worry. If your external will be making text callbacks (or directly calling other XCMDs that might make text callbacks), then you probably want to call XWAllowReEntrancy during your response to an OpenEvent, with both allowSysEvts and allowHCEvts set to true. Otherwise you could lose events. If you are using INLINES to store data in the code of your compiled external, you may want to think twice about it.

on XWAlwaysMoveHigh windowPtr,moveHigh

Moving handles around in memory takes time, and HyperCard will not normally bother to do this if it is not necessary. However, if your external requires large blocks of space in the heap, it may be desirable to have HyperCard move your external off to one side before locking it down and calling you. XWAlwaysMoveHigh does that. There is no particular reason why a *CompileIt!* XCMD should require this service — unless of course you are working with very large data handles. You should try to turn this feature off again by calling XWAlwaysMoveHigh with moveHigh set to false as soon as you no longer need it.

on XWHasInterruptCode windowPtr,lockMe

If your external window has some code running on a VBL interrupt, then you cannot have HyperCard moving your code resource around in memory while you are not looking. Use this callback to let HyperCard know that is the case. Note that this applies only to the time between calls to your external; it is always locked while you are running. Therefore you do not need to use this callback at all if you are passing procPtrs to the Toolbox only when you get an event (use an INLINE to calculate the address of your filter function or userItem each time you get an event from HyperCard, rather than saving and re-using the same pointers). HyperCard gets very disagreeable if you leave too many handles locked for too long (see the section on *Safe Pointers*). Note also that you should

not assume that your external has been locked until the next event *after* you call `XWHasInterruptCode`, since HyperCard will likely try to move you to an out-of-the-way location before locking your handle. Be sure to call `XWHasInterruptCode` again with `lockMe` set to false when you no longer need to remain locked.

```
function ZeroTermHandle(hndl) -- HyperCard string result
```

Insert a terminating null at the end of the handle `hndl`, and return it as a HyperCard string. Use *CompileIt!*'s `HyperCardText` function if the handle already has a null. Use the `HyperCardOwns` command on the resulting string if you don't want *CompileIt!* to take ownership of it.

```
on ZeroToPas zeroStr,pasStr
```

Copies characters from memory starting at the address in `zeroStr` to the P-string variable `pasStr`, stopping at the first null, or when 255 characters have been copied. Note that `zeroStr` is a pointer, not a handle or string.

```
on ZeroBytes dstPtr,longCount
```

Sets `longCount` bytes in memory to zero, beginning at the address in `dstPtr`.

External Window Events

There are three kinds of events that HyperCard sends to external windows and *CompileIt!* manages for you. The first group is designed to initialize and support the debugging tools that HyperCard has delegated to externals. The externals that are of the Script Editor/Debugger variety are opened with a script and a pointer to the descriptor record for the object. A pointer to a P-string with a suggested window name is also supplied. These editors are very closely coupled, and it would be difficult to implement an editor that is much different from HyperCard's. Six more events are used to communicate special information to these windoids.

The remaining events are much more generic and can be sent to any external window. Some of these events are queries or otherwise are required to return some reply to HyperCard; *CompileIt!* defines these as HyperTalk functions, and the reply is simply the result returned by the function. The events that require no reply can either originate within HyperCard as a consequence of some other event, or can be passed directly from the operating system events caught by HyperCard's own event loop. There appears to be no particular advantage to distinguishing them by source this way.

Except for the debugging utility initialization events, all come with a parameter which is the pointer to HyperCard's XWEventInfoPtr block. The two events dealing with xWindoid properties also come with a pointer to the P-string name of the property, and in the case of setting the property, a value string to set it to. Most of the time you do not need any access at all to the XWEventInfoPtr block, but it is available just the same. One of the fields in this data structure is the windowPtr for your window (events are only sent to externals with windows); *CompileIt!* automatically does a SetPort on this window before calling your event handler, so you can easily recover the windowPtr by calling GetPort.

There is an extended example in the **Examples** card that shows how some of these window events can be managed correctly.

```
on InitializeMessageWatcher
on InitializeVariableWatcher
on InitializeScriptEditor theScript,windowNamePtr,TalkObjectPtr
on InitializeDebugger theScript,windowNamePtr,TalkObjectPtr
```

These four events serve to initialize their respective debugging windoids. Note that the Claris Script Language Guide warns against the script editors not being re-entrant. Fortunately this is a relatively difficult mistake to make in *CompileIt!*

```
on ShowWatchInfoEvent eventInfoPtr
on ScriptErrorEvent eventInfoPtr
on DebugErrorEvent eventInfoPtr
on DebugStepEvent eventInfoPtr
on DebugTraceEvent eventInfoPtr
on DebugFinishedEvent eventInfoPtr
```

These six events serve to communicate particular information to their respective debugging windoids.

```
on OpenEvent eventInfoPtr
```

This is the first event sent to an external window after the XCMD that created it completes execution. You should use this opportunity to create any data structures, show your window, etc.

```
on idleEvent eventInfoPtr
```

Your window will get idle events only if you have called `SetXWIdleTime` with a non-zero time in ticks.

```
on keyDownEvent eventInfoPtr
```

```
on autoKeyEvent eventInfoPtr
```

Your window will get `keyDown` and `autoKey` events only if you have called `BeginXWEdit` and not yet received a `GiveUpEditEvent`.

```
on mouseDownEvent eventInfoPtr
```

```
on updateEvent eventInfoPtr
```

```
on activateEvent eventInfoPtr
```

```
on app4Event eventInfoPtr
```

These four events are the same as the corresponding Toolbox events in *Inside Macintosh*. Apple recommends that windoids in the floating layer hide themselves when sent a Suspend event, and show themselves on Resume. A copy of the Toolbox event record can be reached using the expression, `eventInfoPtr@.event.what` (or `eventInfoPtr@.event.where`, etc.).

```
on EditEvent eventInfoPtr
```

The five standard items under the Edit menu are collected together into this one event, more for *CompileIt!*'s convenience than yours. You can determine which event it was by testing,

```
if eventInfoPtr@.what=xEditUndo then -- do Undo event
else if eventInfoPtr@.what=xEditCut then -- do Cut event
```

and so on. Or you can calculate `eventInfoPtr@.what-xEditUndo`, which will give you the values, 0=Undo, 2=Cut, 3=Copy, 4=Paste, 5=Clear, corresponding to the positions of these items in the Edit menu (less 1), or the values used when you call `SystemEdit` from an application event loop (see *Inside Macintosh*, I-441). Your window will only get Edit events if you have called `BeginXWEdit` and not yet received a `GiveUpEditEvent`.

```
on GiveUpEditEvent eventInfoPtr
```

If your window called `BeginXWEdit`, this event lets you know that you are losing it. Unlike `GiveUpSoundEvent`, you have no choice in the matter.

```
on HidePalettesEvent eventInfoPtr
on ShowPalettesEvent eventInfoPtr
```

When another XCMD uses the `HideHCPalettes` or `ShowHCPalettes` callback, the corresponding event is sent to all windows in the floating layer.

```
on SendEvent eventInfoPtr
```

You will get this event if a handler script or another external window is sending your window a message. The text of the message (generally, its name) is in

```
eventInfoPtr@.eventParams.ptrtype@.Str255type.
```

```
on MenuEvent eventInfoPtr
```

Your window will get this event only if you called `RegisterXWMenu` with registering set to true. When you get it, `eventInfoPtr@.eventParams.LongIntType[1]` contains the menuID of the selected menu, and `eventInfoPtr@.eventParams.LongIntType[2]` contains the selected item number. You get the same event whether the menu item was selected by pulling it down with the mouse, or by typing its commandKey equivalent.

```
on MBarClickedEvent eventInfoPtr
```

If you have registered one or more menus using the `RegisterXWMenu` callback, then you will get this event whenever the user clicks on the menuBar, before any menus are dropped down. This allows your external to adjust the menu items of your menus if necessary before the user sees them.

```
function CloseEvent eventInfoPtr -- return true if closing
```

This is normally the last event sent to a window before it is closed and its window disposed — unless the window refuses to close. It is a function so that if you wish to put up a dialog including a **Cancel** button, your window will stay open. Normally you would use this event to dispose all your private data structures (but not the window itself!), then return true to signal that it's OK to close your window. If the user tries to Quit HyperCard and your window returns false to the Close event, HyperCard will be unable to quit. Do not dispose of your data structures until you

receive the Close event: if you have unprocessed or incompletely processed messages pending, their handlers might become confused if the data is already gone. If you do not include a CloseEvent handler in your script, *CompileIt!* will supply a default handler that returns true.

```
function GiveUpSoundEvent eventInfoPtr -- return true if ok
```

This event signals HyperCard's request that you give up the sound channel. You should return true if you are willing to do so, and have terminated all your Sound Manager calls. If you return false, then HyperCard will be unable to make the sounds requested by the script or release the sound channel to another xWindoid as requested. Remember, the other xWindoid that wants the sound channel might be another instance of your own XCMD. If you do not include a GiveUpSoundEvent handler in your script, *CompileIt!* will supply a default handler that returns true, but you really should have a handler for this event if you called BeginXSound, and you wouldn't even get the event if you did not make that callback.

```
function CursorWithinEvent eventInfoPtr
```

This event is something like the mouseWithin message sent to a button or field in HyperCard, except that it is assumed that you might want to set the cursor to reflect what the mouse is pointing to. If you return true, then HyperCard will set the cursor to a default arrow. Return false if you are setting the cursor, and HyperCard won't touch it while it is over the content region of your window. If you do not include a CursorWithinEvent handler in your script, *CompileIt!* will supply a default handler that returns true.

```
function SetPropEvent eventInfoPtr,propNamePtr,propertyValue
```

```
function GetPropEvent eventInfoPtr,propNamePtr -- return value or exit
```

These two events allow your window to service its properties. The propNamePtr for both of these events is a pointer to a P-string with the name of the property to be processed. The property value in each case is a zero-terminated HyperCard string.

An external window may have as many properties as it cares to manage. HyperCard will manage two default properties for any window: loc and visible. Any other property that the external declines to service will result in an error dialog. If the window handler chooses to service a SetPropEvent, it should return false when it has accepted it; if it returns true then HyperCard will apply its default handler. Similarly, the external

window can manage its own replies to the `GetPropEvent` by returning the value of the requested property. Since empty and "false" and "true" are all valid property values, *CompileIt!* recognizes that you want to let HyperCard apply its default handler when you exit this handler without returning any value at all — not even empty, just exit `GetPropEvent` or fall through to the `end GetPropEvent` line without executing any return command. If you do not include a `SetPropEvent` handler in your script, *CompileIt!* will supply a default handler that returns true, and if you do not include a `GetPropEvent` handler in your script, *CompileIt!* will supply a default handler that just exits.

HyperCard Data Structures for Externals

The `XWEventInfoPtr` passed as a parameter with most external window events has the following four fields (shown using the syntax for shared variable data types, even though these are of course not the same as shared variables):

```
event:Record[16] -- a standard event record from Inside Macintosh I-249
eventWindow:Pointer -- a standard windowPtr from Inside Macintosh I-276
eventParams:Record[40] -- an array of ten LongInts
eventResult:Pointer -- the requested property value handle from GetPropEvent
```

Script editing externals need access to the internal object descriptor, which has this format:

```
objectKind:Integer -- 1=stack, 2=bkgnd, 3=card, 4=field, 5=button
stackNum:LongInt -- reference number of the source stack
bkgndID:LongInt -- the bkgnd of the card (if any) of the. . .
cardID:LongInt -- the card where this button or field (if any) resides
buttonID:LongInt -- buttonID is its real ID, bkgndID = NIL for card objects
fieldID:LongInt -- fieldID is its real ID, bkgndID = NIL for card objects
```

The `XcmdBlock` is passed by a pointer to the external for all calls, including external window events. Note that *CompileIt!* manages this block for you. Its format:

```
paramCount:Integer -- negative if an event
params:Record[64] -- an array of 16 LongInts or pointers;
    -- for events, params.ptrtype[1] contains the XWEventInfoPtr
returnValue:Pointer -- really a zero-terminated handle, filled when
    -- you return
passFlag:Boolean -- true when you pass handlername
entryPoint:Pointer -- callbacks jump to this address
request:Integer -- the callback request number (shown in the
--Callback Review dialog)
resultX:Integer -- 0=OK, 1=failed, 2=not implemented, usually in
    -- "the result"
inArgs:Record[32] -- an array of 8 LongInts or pointers or handles,
    -- for callbacks
outArgs:Record[16] -- an array of 4 LongInts or pointers or handles
```



INDEX

Symbols

\$ 42, 53
@ 75, 79, 85, 93
(...) 79

A

Add All button 58
AddResource 96
Address 79
Always Install in Stack... 22
Always Use SANE 25
America Online 57
Analysis 45
Analysis button 36, 53
Analysis card 46, 52, 53
Array notation 74, 78, 87, 89, 98, 101
Assembly language 11

B

Batch compile 22
Binary callbacks 32, 35, 46, 66, 71
BitMap 30, 89
Bkpt 46
BlockMove 94
Boolean 28, 70, 89
Breakpoint 46, 49, 91

C

Callbacks 34, 51, 62
Char 89
Character strings 77, 94
Characters 82
CharsHandle function 60, 75, 78, 82, 93, 96
Chunk expression 76, 86, 91
Clear Marks button 60
CloseDialog 109
CloseEvent 107

CloseWindow 109
CloseXWindow 109
Clr Bkpts button 49
Command button 50
Command-period 49, 51
Compilation phase 21
Compile It button 18
Compiler 11
CompuServe 57
Coordinates 71
Copy & Rename button 65, 73
Custom Symbol Edit card 59, 63, 73, 87

D

Data type 27, 47, 70, 100
DebugIt! 19, 42, 90
DebugIt! checkbox 42
DebugIt! option 42
Demo 17
Dereference operator 75, 85, 87, 93
Desk accessories 22
Desk Accessory names 29
DetachResource 96
Diamond marker 47, 60
DisposDialog 109
DisposeWindow 109
DisposPtr 80, 88
Dollar sign 53
Dot 30, 86
Duplicate symbols 73

E

Edit Hex button 47
Edit Marked 66
EditEvent 107
EntryPoint 110
Error detection 42
EvalExpr 43
Event record 86
EventAvail 86
EventInfo parameter block 108

eventParams array 109
EventRecord 89
EventResult 110
Exit to HyperCard 19
External commands 12
External windows 96, 105
ExtToStr 110

F

"Fast" mode 43
Field 37
Fields 26, 71, 86
Fixed 89
Floating point 25, 66
FrontDocWindow 52
FrontWindow 52
Functions 41

G

GetFieldByName 110
GetFieldByNum 110
GetFieldTE 111
GetGlobal 114
GetNewXWindow 109
GetNextEvent 87
GetPropEvent 107
GetPtrSize 89
GetResource 96
GetXCmdPtr 115
Glo/Fld 46
Global variables 38
GrafPort 89

H

HandAndHand 86
Handle 41, 47, 60, 75, 77, 79, 81, 82, 85, 89, 90
Handle Chunks 82, 84, 98
HandToHand 96
heap 79
help 16
HiWord 39

HLock 85
Host application 43
Hour hand 21
HUnlock 85
HyperCard 1.x XCMD interface 43, 50, 52
HyperCard 2 checkbox 19, 49
HyperCard 2.0 12, 44, 46, 62, 71, 105
HyperCard 2.0 Extended XCMD Interface 105
HyperCard string 29
HyperCardOwns 96, 111
HyperCardText 96, 112
HyperDebugIt 43
HyperTalk names 60

I

Incremental compiler 12
Inline jump 64
Inside Macintosh 30, 40, 59, 69, 86-87, 89
Integer 28, 71, 82, 89
Integer arithmetic 25
IntegerType 73
It 30, 47, 48
ItemDel property 49
ItemDelim property 49
ItemDelimiter property 49

L

Library Code 32, 33
Licensing 53
Long Integer 71
LongInteger 47
LongInts 82, 89
LongIntType 73
Low-Memory Global 70, 84
LoWord 39

M

MacsBug 52
Mark Only Dupes button 60
Marked Used Names 59
Master pointer 81, 85

MemErr 84
Memory 79
Minute hand 21
Motorola MC68000 11
MouseUp 14
Munger 94, 98

N

Native Inline Code 32
New Name... button 63
NewHandle 96, 101
NewPtr 80, 87, 101
NewXWindow 109
Non-existent chunk 82
Non-printable string characters 47
Non-printing control character 49
Null characters 29, 95
NumCvt 46

O

OpenEvent 107
Optional parameters 65
Options card 22, 43
OSType 73, 89
Other Options... button 57

P

P-strings 29, 99, 101
Palette 106
Paramcount 65
ParamCountX 112
Parameters 13, 18, 69, 96
ParamPtr 109
Parentheses 41
Paste from Clipboard button 17
PasToZero 113
Pattern 89
Point 71, 88, 89
Pointer 41, 47, 75, 77, 79, 81, 82, 85
Pointer arithmetic 85, 87, 98, 102
Pointer chunks 83

PrintTEHandle 113
Procedures 41
Programming errors 42
Progress 20
Pseudo fields 74, 88
Ptr 89

R

Random function 60
Record 29, 47, 101
Record field notation 86
Record structure 71
Rect 72, 89
Rectangle 72
reentrancy 105
RefCon 112
Refresh 43
Refresh button 50
Remove All button 58
Remove Marked button 60
Reserved handler names 105
Resize 51
Resource Manager 96
ResultX 113
Resume button 49
ReturnToPas 113
returnValue 113
Review Binary Callbacks 51
Review Text Callbacks 51
ROM Toolbox routines 29
ROM/Libc 46, 86
runtime fees 53

S

SANE 25, 28, 89
SaneType 73
ScanToZero 113
Script card 43
Script card 20
Send Msg 50
SendCardMessage 43

SendHCmessage 43
SetFieldByID 110
SetFieldByName 110
SetFieldByNum 110
SetFieldTE 111
SetGlobal 114
SetHandleSize 86, 95
SetPropEvent 107
SetPtrSize 81, 89
SFGetFile 102
Shared variable 72
SignedByte 89
Size of DebugIt! 52
Small stack icon 22
Source level debugger 42
Standard Apple Numeric Environment 19, 25, 28, 89
Step button 49
Stop Sign 49, 51
STR# 99
Str255Type 73
String 29
String compares 76
String handling 75
StringEqual 114
StringLength 114
StringMatch 114
Strings 66
StrToBool 114
StrToExt 114
StrToLong 115
StrToNum 115
SuperCard 57, 62, 65
Switch windows 51
Symbol Manager 58
Symbol table 57
System globals 80

T

Text callbacks 32, 46, 54, 66
the result 48
Ticks 62

TMON 52
Toolbox 40, 69, 75
Toolbox Function 70
Toolbox Global 70, 84
Toolbox procedure 69
Trace 49
TxtCbK 46

U

UpdateEvent 107
Updater stacks 59

V

Value button 50
Value function 62
VAR 70
Variable monitor 47

W

WaitNextEvent 87
Watch cursor 21
Within 36

X

XCMD 12
XCmdPtr 115
XFCN 12
xWindoids 105

Z

ZeroToPas 115
