Practical XFCNs: Free HyperCard Utilities

By Ari Halberstadt

ABSTRACT

Information common to several general purpose external functions for HyperCard (on the Macintosh): ordered array, binary tree, and a regular expression searcher. A comprehensive manual and source code in C is included for each program. The programs are free; for distribution terms see the appropriate sections in this manual.

This manual is intended for people who write scripts for HyperCard.

Table of Contents

Sections

Tables

Scripts

Script 2. Using ErrorName

Introduction

This file describes features that are common to at least three external functions (XFCNs) which I have written for HyperCard. The external functions are: ArrayList, an implementation of an ordered list stored as an array; BinaryTree, an implementation of all basic operations on a binary tree; and Research, a regular expression searcher, with numerous options to modify its behavior. Each of these external functions is free and is distributed with its source code in C. The programs are not, however, in the public domain, and there are some restrictions on their distribution which are described in the appropriate sections in this manual.

Each program has its own comprehensive manual, which you should read for details. This manual was written so that I would not repeat the same things such as installation details and error recovery in each of the manuals.

I would like to thank Carol S. and Jerome Halberstadt for their helpful suggestions.

Installation

To install the programs you must copy several resources into your stack using ResEdit[1] or a resource copier XFCN. These resources consist of the executable programs, and other resources which contain data needed by the programs. A list of the resources needed by each program is given in an appendix accompanying the manual for the appropriate program.

All of the resources are contained in the demonstration stack supplied with the programs. To install a specific program, copy the resources listed in its manual into your stack.

During installation, there is a slight chance that other resources already installed in a stack have the same ID as one or more of the resources used by these programs. If this problem occurs, please read the section in this manual on solving resource ID conflicts before proceeding with the installation (you will need a program such as ResEdit to fix resource ID conflicts).

[1]ResEdit is an application available from Apple for editing resources.

Using the programs

Assuming you have successfully installed the programs into your stacks, this section describes some features common to using each of the programs.

Each program may execute different internal functions depending on the parameters supplied to the XFCN. A typical way to call one of these XFCNs is,

```
get btree(function, parameters)
```

In addition, each program implements several standard functions, each of which is briefly described below (the precise way to call these functions and the data they return should still be looked up in the relevant manual).

Syntax descriptions

The manual for each program gives complete descriptions of the functions it implements. The syntax descriptions for the functions use the following typographic conventions[2]. Words or phrases in `typewriter` type are to be typed to the computer literally, exactly as shown. Words in *`italic`* type describe general elements, not specific names — you must substitute the actual instances. Square ([]) brackets enclose optional elements that may be included if you need them. (Don't type the square brackets.)

Some function descriptions have been simplified so as to avoid excessive detail about the internal operation of the program. The program may in fact use more efficient algorithms for some operations than those described.

Return types

Each function description is preceded by a word indicating the type of data that the function returns. This word is purely for your information and should not be entered in HyperTalk scripts. Some functions may act as procedures, returning the empty string. The following table lists the basic data types passed to and returned from the programs.

Table 1. **Data types**

Type	Description
Boolean	The string "true" or "false".
error	An error code. Value is empty to indicate no error, otherwise it

[2]Adapted from “HyperCard Script Language Guide: The HyperTalk Language”, Addison-Wesley (1988).

is either an integer or an error name. Error codes are described in more detail later in this manual.

separator — A single character used to delimit substrings. Separator characters always match exactly; that is, comparisons between a separator character and another character always take the capitalization of letters into account (for instance, lower case a is not considered the same as upper case A). HyperCard's item separator, which is a comma, is usually the default separator used when no other separator is specified.

string — Any HyperTalk string.

empty — Nothing is returned.

Common functions

This section describes functions which are provided by most of the programs. You should consult the manual for the relevant program for more precise details about these functions.

"!"

Syntax

string alist("!")
string btree("!")
string research("!")
string errorstring("!")

```
string errorname("!")
```

Description

Returns a string describing the version of the program, along with a copyright statement, the name of the author, and the date and time of compilation. If two copies of the same program have the same version numbers, but different times of compilation, then the one with the more recent compilation time should be used.

"?"

Syntax

string alist("?")
string btree("?")
string research("?")
string errorstring("?")

```
string errorname("?")
```

Description

Returns a string giving a brief summary of the ways to call the XFCN. The string may extend to more than one line, so it may be helpful to display it in a scrolling field. In the case of the more complicated XFCNs, the string may also simply refer you to the manuals.

Error

string alist(error)
string btree(error)

Syntax

```
string research(error)
```

Description

Returns the error code of the last error that occurred, or empty if the last function executed successfully. Should be called after each call to the program that does not return an error code. You can use the ErrorString and ErrorName XFCNs (described later in this manual) to get a string describing the error that occurred.

Notes

This function may, in a future release, be expanded to provide the syntax and a brief description for each individual command. For instance, when called as

```
btree("?", inorder)
```

a string describing the Inorder function of the BinaryTree XFCN would be returned.

Errors

Every function executed from the programs, except for the Error function itself, may encounter an error. There are many things that may cause an error. For instance, you may call the program incorrectly, or the program may run out of memory, or the program may be unable to locate a resource. It is always a good idea to check for errors. This helps make your scripts more robust and ensures that they will always do something reasonable and helpful.

Two external functions are provided for accessing the error codes: ErrorName and ErrorString. The following sections show how to use these external functions to check for and recover from errors.

Checking for errors

Each program has a special function called Error for getting at the last error that the program encountered. This function returns an integer, called an **error code**, which identifies the error that occurred. If no error occurred then the Error function returns the result empty. A typical way to call one of the programs is:

```
get alist(new, array) -- create a new array
if (alist(error) <> empty) then reportError alist(error)
```

In this example, we call ArrayList, and then check if an error occurred. We report any errors to the user and exit (the handler reportError is shown below). Some functions return an error code, so we don't even have to use the Error function. The function New of ArrayList returns an error code, or empty when it's successful, so that we can rewrite the previous example to be more efficient:

```
get alist(new, array)
if (it <> empty) then reportError it
```

This second version saves us the overhead of an extra call to ArrayList.

Reporting errors

The error codes returned by the programs are integers. Negative error codes are returned when a Macintosh system call fails[3], and positive error codes are returned when a part of the program itself fails. The ErrorString XFCN is used to get a short

[3]This "fact" was determined empirically, based on experience programming the Macintosh. I do not recall actually seeing such a promise from Apple.

description of an error, which can then be used to explain to the user what went wrong. For instance, the following script will report errors

to the user.

Script 1. **ReportError**

```
on reportError errorCode
      if (errorCode <> empty) then
            answer "Error:" && ErrorString(errorCode)
            exit to HyperCard
      end if
end reportError
```

Notice that ErrorString only contains strings describing errors produced by the other programs in this set (eg, ArrayList, BinaryTree, Research). These programs may also encounter errors caused by the failure of a Macintosh system call. If the error code is negative, then it refers to a Macintosh system error, and ErrorString will simply return the error code itself, instead of a descriptive string. For instance, the command

```
get ErrorString(-108)
```

places the string "-108" into the variable it. (This is the error code returned when the Macintosh has run out of memory.)

The strings returned by ErrorString are summarized in a table in a following section.

Checking for specific errors

Occasionally you may need to know exactly which error occurred, so that your script can handle certain errors in specific ways. The programs return errors in the form of integers. However, when referring to a specific error, you should not use the integer value, rather you should refer to the error's name. An **error name** is a single unique word which identifies the error. You can convert an error number into an error name using the ErrorName XFCN. For instance,

```
ErrorName(6) -- returns "ERR_BASIC_HYPERCARD"
```

The ErrorString XFCN will also convert error names into descriptive strings, so you can call ErrorString even if you've only got the error's name. For instance:

```
ErrorString(6) -- returns "HyperCard error"
ErrorString(ERR_BASIC_HYPERCARD) -- returns "HyperCard error"
ErrorString(ErrorName(6)) -- returns "HyperCard error"
```

If ErrorName doesn't recognize the error number, then it returns whatever its parameter is. For instance,

```
ErrorName(ERR_BASIC_HYPERCARD) --returns "ERR_BASIC_HYPERCARD"
ErrorName(-108) -- returns -108
ErrorName("Return Me") -- returns "Return Me"
```

Notice that ErrorName only contains names of errors produced by the other programs in this set (e.g., ArrayList, BinaryTree, Research). These programs may also encounter errors caused by the failure of a Macintosh system call. If the error code is negative, then it refers to a Macintosh system error, and ErrorName will simply return the error code itself, instead of its name. For instance, the command

```
get ErrorName(-108)
```

places the string "-108" into the HyperTalk variable it. (This is the error code returned when the Macintosh has run out of memory.) You should always use ErrorName to get the error's name; if ErrorName didn't recognize the error, then you can refer to the actual error number, otherwise you should refer to the error's name.

As an example, suppose you use ArrayList to maintain lists created by the user of your stack. If the user were to attempt to create a new list, but the list already existed, you might want to display a dialog box asking whether to replace the old list with the new list. The following script shows how you could do this.

Script 2. **Using ErrorName**

```
on newList listName
      get alist(new, listName) -- create list
      if (it <> empty) then -- couldn't create it
            get ErrorName(it) -- get name of error
            if (it = "ERR_LTABLE_EXISTS") then --check if list
exists
                  -- query user
                  answer "Replace existing list " &¬
                  quote & listName & quote & "?" with "Yes" or "No"
                  if (it = "No" or it = empty) then exit newList
                  -- dispose of old list and create new list
                  get alist(dispose, listName) -- dispose of list
                  if (it <> empty) then reportError it
                  get alist(new, listName) -- create list
                  if (it <> empty) then reportError it
            end if
      end if
end newList
```

Error names and descriptions

The following table lists the names and the descriptions of the errors. You can access the names of the errors using the ErrorName XFCN, and the descriptions of the errors using the ErrorString XFCN.

Table 2. **Errors**

Returned By	Name	Description
All Programs	ERR_BASIC_UNKNOWN	"Unknown error"
	ERR_BASIC_PROGRAM	"Program error"
	ERR_BASIC_ROMVER	"Wrong ROM"
	ERR_BASIC_SYSVER	"Wrong system"
	ERR_BASIC_HCVER	"Wrong HyperCard"
	ERR_BASIC_HYPERCARD	"HyperCard error"
	ERR_BASIC_ARGCNT	"Parameter count"
	ERR_BASIC_USAGE	"Parameter usage"
	ERR_BASIC_COMMAND	"Unknown command"
	ERR_BASIC_UNIMPLEMENTED	"Unimplemented command"
	ERR_BASIC_CANCELED	"Operation was canceled"
ArrayList	ERR_ARRAY_INDEX	"Subscript out of bounds"
	ERR_ARRAY_NEGCNT	"Negative count"
	ERR_ARRAY_LIST	"Unused error code"
	ERR_ARRAY_EMPTY	"Empty array"
	ERR_ARRAY_ATTR	"Unknown attribute"
	ERR_ARRAY_EXISTS	"Array already exists"
	ERR_ARRAY_NOT_EXISTS	"Array doesn't exist"
	ERR_ARRAY_SUBSCRIPT	"Illegal subscript"
	ERR_ARRAY_CHKSUM	"ArrayList checksum error"
Research	ERR_PAT_EOF	"Pattern ended unexpectedly"
	ERR_PAT_LITERAL	"Missing literal in pattern"
	ERR_PAT_DIGIT	"Missing digit in pattern"
	ERR_PAT_ENDTAG	"Missing '\)' in pattern"
	ERR_PAT_ECCL	"Missing ']' in pattern"
	ERR_PAT_EGROUP	"Missing ')' in pattern"
	ERR_PAT_MAXTAG	"Too many tags in pattern"
	ERR_PAT_SIZE	"Pattern is too big"
	ERR_PAT_ENDCLOSURE	"Missing '}' in pattern"
	ERR_PAT_MAXMIN	"Closure's min is greater than max"
	ERR_PAT_MATCH	"Pattern is too complex for input"
	ERR_RESEARCH_OPTION	"Unknown option"
	ERR_RESEARCH_CHKSUM	"Research checksum error"
BinaryTree	ERR_TREE_EXISTS	"Key already exists"
	ERR_TREE_NOTFOUND	"Key doesn't exist"
	ERR_TREE_ATTR	"Unknown attribute"
	ERR_TREE_EMPTYKEY	"Empty key"

ERR_TREE_EXISTS	"Tree already exists"
ERR_TREE_NOT_EXISTS	"Tree doesn't exist"
ERR_TREE_CHKSUM	"BinaryTree checksum error"

Resources used by ErrorString and ErrorName

The following table lists the resources required by the ErrorString XFCN. These resources must be installed in your stack for ErrorString to work.

Table 3. **Resources used by ErrorString**

Type	Name	Description
XFCN	ErrorString	The ErrorString XFCN.
TABL	ErrorString:Strings	Descriptions of the errors.
STR#	ErrorString:ResourceMap	List of resources used by ErrorString.
STR#	ErrorString:Info	Version and usage information for ErrorString.

The following table lists the resources required by the ErrorName XFCN. These resources must be installed in your stack for ErrorName to work.

Table 4. **Resources used by ErrorName**

Type	Name	Description
XFCN	ErrorName	The ErrorName XFCN.
TABL	ErrorName:Names*	Names of the errors.
STR#	ErrorName:ResourceMap	List of resources used by ErrorName.
STR#	ErrorName:Info	Version and usage information for ErrorName.

* Resource is also used by ErrorString.

HyperTalk global variables

This section lists the global variables used by the programs. This information is provided so that you do not use the same global variables in your scripts, and for documentation purposes for the author of these programs.

Note: You should not access these global variables since they may cease to exist in future versions. Also, modifying the contents of these variables may lead to disastrous results (such as system bombs).

Declaring

Some older versions of HyperCard seem to require that global variables be declared somewhere in a script before the variables can be used. If you're using one of those versions [earlier than 1.2(?)], then you should include lines like the following ones in an openStack handler:

```
global _ArrayListGlobal
global _BinaryTreeGlobal
global _ResearchGlobal
```

You should not refer to these global variables anywhere else in your scripts.

Globals

The following table lists the global variables needed by the programs.

Table 5. **HyperTalk global variables**

Variable	Description
_ArrayListGlobal	Handle to data used by ArrayList
_BinaryTreeGlobal	Handle to data used by BinaryTree
_ResearchGlobal	Handle to data used by Research

About the source code

This section is intended for programmers interested in understanding and/or modifying the programs. This section contains general information about the programs; for more specific information read the appropriate section in each program's manual.

You should know how to use the C programming language (in which I wrote the programs) on the Macintosh, and a knowledge of how to write XFCNs will be helpful. Though the programs were written using THINK C 4.0, it should be fairly straightforward to adapt them for other development environments, and I've included a section that will be helpful for those porting to MPW C. When viewing the source code you should use 3 spaces per tab.

The code for these programs was surprisingly large when I first compiled them. Evidently, my passion for highly structured code and strict error recovery results in programs which are quite large. I would, however, like to think that my programs are more understandable, maintainable, and correct than less strict programs. Anyway, most inefficiencies in my code should eventually be removed by better optimizing compilers and size problems should become insignificant as computer memories grow larger and cheaper. I remember reading in one of my textbooks that a programmer should not fiddle with code to make it faster, rather, a better algorithm should be used (though I admit I did, at times, "fiddle" in order to get improvements...).

They're really PROC resources

Since speed was such a crucial consideration, and knowing that the programs may be called many times, I had to figure out a way to limit the time HyperCard spends loading the programs into memory. As you probably know, HyperCard makes a complete copy of an XFCN resource before jumping to it. The solution was to write a short XFCN whose sole purpose is to load a PROC resource, lock it in memory, jump to it, and then unlock the PROC resource and return. The PROC resource is where the true "XFCN" is. I always refer to the programs as XFCNs, and not as PROC resources, since I prefer the terminology and because the functionality is nearly identical[4].

[4]Actually, there is a slight difference. If the stack containing the PROC resource is closed while it is executing then the currently executing code may be lost, resulting in indeterminate behavior.

Source code organization

Folder organization

Source code which is used only by a single program is contained in a folder with a name such as "TreePROC" for the BinaryTree program. Source code which is shared by all programs is contained in the folder named "Libraries". The libraries folder contains a folder for each individual library, which in turn contains the source file and a header file containing definitions for that library. Some of these libraries are compiled into a single library, called "Libs-A4.π", which is loaded into each of the programs. Since the full ANSI library is not needed, this library uses the code from only a few of the source files for the ANSI library, thus resulting in smaller projects and faster compilation.

Projects

Each program is composed of two projects[5]. One is the project for the PROC resource, and the second is the project for the small XFCN. Both must be compiled in order to run the program. Notice that the XFCN project does not include THINK C's MacTraps library. This results in a much smaller program, and is possible since I wrote the functions for loading the PROC resource in assembly language.

Compiling

To compile the projects you must follow several steps:

1. You should move the "Libraries" folder into the same folder as the THINK C application. This will allow all of the projects to access the files in the libraries.

2. To speed up compilation during development, I defined my own precompiled header file, instead of using the larger MacHeaders file supplied with THINK C. Therefore, every ".c" source code file starts with the line "#include <MyHeaders>". The file "MyHeaders.c" contains the include statements, which, when compiled using the precompile command, will yield the file "MyHeaders". You should precompile this file and place it in the THINK C folder before compiling any of the programs.

3. Once the file "MyHeaders" is ready, build the project "Libs-A4.π" as a code resource. The project should have access to the source files for the ANSI library provided with THINK C. Ignore any link errors, since this project is only being used as a library.

4. Now, compile the project for the XFCN part of the program (it will be named "TreeXFCN.π", or something similar). Finally, compile the project for the PROC part of the program (it will be named "TreePROC.π", or something similar).

[5]THINK C uses special files called "projects" to keep track of the source files used in a program.

Profile option

Be careful if compiling with the profile option. The assembly language routines that I wrote fail due to the extra profiling code inserted by the THINK C compiler. Therefore, compile files containing assembly language with the profile option turned off, then turn the option on and compile all the other files. Alternately, you can simply turn assembly language off by setting the appropriate flag in the configuration header file ("stdconfig.h"). I don't know what effect the profile option would have in MPW.

Compile options

Several preprocessor symbols are assigned default values in "stdconfig.h"; in addition, every project has a file named "privateconfig.h" which can be used to override any of the defaults. Following are descriptions of what some of the flags do.

DEBUG

If compiled with DEBUG defined as 1 then some debugging code will become active. Most important, if the shift key is held down when execution of a program is starting then the debugger trap is executed; you should have a low level debugger such as MacsBug or TMON if you hold down the shift key.

ASSERTIONS

If ASSERTIONS is defined as 1 then assertions will be activated, which aids greatly in the debugging process. Also, some operations will be implemented as functions instead of equivalent macro definitions. The effect of this is to slow down the programs and to increase their size by several thousand bytes. ArrayList got so big I had to compile it in two segments to test it using assertions.

CUSTOM_HEADER

THINK C versions 3.0 and up allow the use of a header file other than the default one for code resources. When this option is checked, the THINK C compiler does not insert glue code at the head of the code resource, and instead places the first function in the file with the main routine at the head of the code resource. You must define CUSTOM_HEADER as 1 if this option is used, and you should modify the special header function in "codelib.c". The custom header option will modify the way the program gets a handle to the actual PROC resource. **Note:** This feature has not been tested.

Special assumptions and comments

Moving memory

I have assumed that calls to standard C library string functions such as strcpy, strcat, and strcmp and also calls to the functions PtoCstr and CtoPstr and to some of the functions in my own string library (such as substrlen and substrcmp) do not move or purge memory. Therefore, I do not lock handles before dereferencing and passing them as parameters to these functions. This would only be a problem if the string

library is moved into a segment other than the main segment (multiple segment code resources are possible in THINK C). When the other segment is loaded into memory the heap may be rearranged. Therefore, always compile the strings library, MacTraps, and the files "strlib.c" and "strasm.c" in the main segment (since the string library is compiled as part of the "Libs-A4.π" project, simply put the Libs-A4 project in the main segment).

Moving object code

The programs are stored in code resources that are loaded into memory while executing and which may be purged from memory while inactive. This means that each code resource may reside at any address in memory, and that the addresses of functions may change from one execution of a code resource to the next. Therefore, any reference to such addresses must be reinitialized each time a code resource is executed. You will notice that there are functions with names like "str_tupdate" (from BinaryTree), and "str_lupdate" (from ArrayList) which do this updating for each data structure.

32-bit

As noted in the requirements section, this program should work in 32-bit clean environments. However, I have written a function called validhandle which helps verify that handles are not corrupt. This function increases the reliability of the software and helps detect bugs. In fact, I think it is one of the most useful functions I've written for debugging programs on the Macintosh. Unfortunately, validhandle uses tests based upon information found in Inside Macintosh, volume II, which may not be valid for new system software. In fact, Apple warns that programmers should not access a handle's header. Therefore, this function may require modification for 32-bit clean environments. (The function validhandle is only used while debugging.)

Testing

Simulating HyperCard

I have included several libraries of code that will be useful for testing changes to any of the XFCNs. These libraries—by simulating a subset of the HyperCard environment—allow you to run the XFCN as an application, thus greatly simplifying debugging. The file "hypersimlib.c" currently implements global variables and translation of Boolean strings. The file "test.c" is a generic testing utility which reads in commands and constructs parameter blocks from them.

Test projects

I have included at least one project (called "TreeTest.π") which can be used to test one of the XFCNs. Other projects can be constructed in a similar manner to test any of the other XFCNs. Notice that, in order to maintain the global variables, "hypersimlib.c" makes use of the library "listlib", so you must include the file "listlib.c" in all test projects. You should also set the preprocessor symbol "APPLICATION" to 1, and the symbol "CODE_RESOURCE" to 0 (in the file "privateconfig.h" for the test project). To make a test project for the Research XFCN, you will have to simulate the EvalExpr call-back, perhaps by returning dummy values.

These test projects are provided as an extra bonus, and I do not intend to expend much effort to maintain them. I usually bring them up to date only when I make extensive changes to one of the XFCNs; in the interim, they may develop inconsistencies. Still, these notes and the comments in the relevant files should be enough to get you started.

Porting to MPW

THINK C and MPW C are quite similar. I have not compiled the programs with MPW, but expect porting them to MPW to be straightforward. You should define MPW as 1 and THINK_C as 0 before compiling (see the file "stdconfig.h"). Following are the main points on which I have found THINK C and MPW to differ:

- THINK C organizes files in projects (those files with the ".π" suffix). In MPW you must create a Makefile describing all the files.

- The names of the header files for using the Toolbox differ. If you're using THINK C version 3.0 or newer you can completely avoid the use of the Macintosh interface header files by using a precompiled header. For MPW, you'll have to explicitly specify which files to include.

- All of the programs share a standard configuration file called "stdconfig.h". In addition, each program requires its own private configuration file, called "privateconfig.h". Each THINK C project therefore has a special folder which is the same as the name of the project, but surrounded with parenthesis. This folder shields its contents, so that only one project can access them. For instance, the project "TreePROC.π" uses the folder "(TreePROC.π)", and one version of the file "privateconfig.h" resides in this folder. MPW doesn't use THINK C's method of shielding folders, but instead uses full pathnames. You'll have to specify a search path to the appropriate "privateconfig.h" file, perhaps by changing folders MPW searches in for included files.

- A few routines were written in assembly language to speed execution. You'll have to figure out how to use assembly language with MPW, or you can define the preprocessor symbol "ASSEMBLY" as 0 (in the file "stdconfig.h").

- THINK C inserts special glue code at the start of a code resource that jumps to the main routine, which may be located anywhere in the program. I believe MPW requires that the main routine be the first function encountered by the

linker. The file "codelib.c" contains the main function for these programs, and must be modified for MPW.

- Though THINK C versions 3.0 and later support global variables in code resources I have made very little use of this feature. The file "codelib.c", which is used to initialize the environment for the code resource, contains a variable used to store a jump-buffer, which allows the programs to abort if they encounter an error; this variable could be removed with no loss in functionality. The functions "RememberA4()", "SetUpA4()" and "RestoreA4()" are used by THINK C to maintain global variables in a code resource. Finally, several files contain string constants, and I'm not sure if MPW will allow them. The solution to this latter problem is to move the strings into a resource.

If you successfully port any of the programs to MPW, I would advise marking the differences with #ifdef statements. I would appreciate receiving news of a successful port so that I may include it in future releases of this software.

Resources

This section describes special resources, building the resources, and solving resource conflicts when installing the programs into a stack.

The TABL resources

The programs use a new type of resource that I defined, called 'TABL', which is short for "Table". The resource is used to look up numbers assigned to a certain string, or to look up a string assigned to a certain number. The routines for accessing this resource type are in "lookuptabl.c" and for building the resource in "buildtabl.c".

Note: The capitalization of letters is ignored when a string is searched for and when the entries in the table are sorted by their strings (letters with diacritics are still compared using regular ASCII ordering).

Format

The first two bytes of the resource give the number of strings, the next two bytes give the width of each text string (must be a multiple of two), and then there are two bytes of flags. The following data are the entries in the table; each entry is preceded by the number of a string, which may be any number, and this is followed by a string padded to the uniform width with zeros. According to the settings of the flag bytes, the table is sorted either by the numbers of the strings or by the strings themselves, so that data may be located efficiently using binary searches.

The resource consists of a header:

2 bytes **Number of entries in table**
2 bytes **Width of each string in table; must be an even number**

2 bytes Flags

This header is followed by the entries in the table:

2 bytes **String number (can be any value)**

Width bytes String padded to width bytes with zeros

Building the TABL resources

The TABL resources are used for looking up such things as the command strings for the XFCNs and then translating these strings into an internal representation. To build these resources, and to keep the resources consistent with the programs, I had to create a little utility program, called "MakeRsrcs.π".

Each XFCN has its own data file, which contains data for the TABL resources that XFCN uses. For instance, the BinaryTree XFCN has a file called "BinaryTreeRsrcs.c". If a command is added or is changed in any of the XFCNs, then you must update the appropriate data files for the program, and then compile and run the MakeRsrcs project. MakeRsrcs will build the TABL resources into the resource file of each program it knows about. You can then compile the program, and it will incorporate the new resources. (The project file "MakeRsrcs.π" should be moved out of its folder, and should be higher up in the directory heirarchy, so that it will have access to the files of the other programs.)

Solving resource conflicts

This section explains how to solve problems caused by a conflict between the resources used by the programs and resources that already exist in a stack. This section assumes some understanding of how the Macintosh handles resources. While reading this section it is a good idea to look at the appendix that gives a list of the resources for the program of interest.

Resources

Every resource on the Macintosh has a type, such as XFCN, STR#, or TEXT, which specifies the type of data stored in the resource. An XFCN resource stores

executable code, a STR# resource stores a list of strings, and a TEXT resource stores plain text. A resource is referred to either by a name or by a number (the number is known as the resource's ID). All resources of a

specific type must have a unique ID, while a unique name is, while not required, a good thing to have. For a combination of reasons (including speed) my programs refer to all resources they need by ID only. HyperCard, however, loads an XFCN by referring to its name, and not its ID. This is necessary since HyperCard doesn't know the IDs of XFCN resources installed in a stack.

Note: Though the resources are installed in a stack, they may still conflict with resources used by the HyperCard application. Therefore, you must be careful that the resource numbers you assign do not conflict with any resources of the same type used by HyperCard.

Loading resources

Most of the resources are loaded by their names, so that it doesn't matter what ID they have. The only exception to this rule are several special resources which are used as maps to the other resources used by the programs. The map resources are of type STR#, and contain the types and the names of the resources used by the programs. These resources **must** have the same ID as the PROC resource to which they belong. Thus, if for some reason you must change the ID of a PROC resource, then you must also remember to change the ID of its map resource.

The following table shows the map resources that must have the same ID as their corresponding PROC resource. The table also gives examples of alternative numberings for the resources (remember to renumber both the PROC resource and the map resource).

Table 6. **Alternative numbering of map resources**

PROC name	Map name	ID	Alternative ID
ArrayList	ArrayList:ResourceMap	2300	3141
Research	Research:ResourceMap	2400	5926
BinaryTree	BinaryTree:ResourceMap	2500	5358

If you've changed any resource IDs. When the XFCNs are first executed, they search for all of the resources they need, and then store the resources' IDs, so that subsequent loading of the resources will be faster. The list of resource to search for is contained in the special map resource. If you've changed any of the resources' IDs, then you'll need to quit HyperCard and then restart HyperCard. This completely erases any data the XFCNs might have stored about the locations of the resources. [Forcing people to quit HyperCard is somewhat annoying, and I will eventually figure out a good way to avoid this problem.]

Loading the PROC resources

Each program is actually stored as a simple XFCN resource that is called from HyperCard and which then proceeds to load and execute a resource of type PROC. The PROC resource is where the executable program resides. The small XFCN loads the PROC resource using a hard-coded resource name. If the resource can't be found then the XFCN exits without doing anything.

Changing resource names

You will probably never need to do this. If you must, then you may have to recompile some of the programs. Specifically, if you rename:

- any of the PROC resources then you will have to recompile its corresponding XFCN;
- the resource maps of the ErrorName or ErrorString XFCNs, then you willl have to recompile either the ErrorName or the ErrorString XFCNs (depending on which ones resources you changed).

Before compiling, you will have to change the names of the affected resources in the appropriate source code files. You should also update the map resources to reflect any changes in the resource names.

Version information

The manuals for each of the programs provide detailed descriptions of any changes from earlier versions.

Experimental version

This manual describes version 0.9 of the programs. The version is numbered 0.9 to indicate that it is an experimental version. I have used the programs somewhat in my own stacks and have verified to my satisfaction that they are correct and that there are no bugs. However, even the best programmer may overlook things. It is very likely that there are some undetected bugs that I hope will be discovered and corrected before the first "real" version which will be numbered 1.0.

The features in this version are somewhat tentative, and I may add or remove features as I see fit prior to version 1.0. With versions 1.0 and up I intend (though I don't guarantee) to maintain backward compatibility with lower versions. Thus, scripts written for version 0.9 may have to be revised slightly to work with version 1.0. Scripts written for version 1.0 should work with all subsequent versions.

Your comments

If you have any comments regarding anything about these programs please don't hesitate to contact me. I'll especially appreciate reports of bugs (though I hope none are found). If you make enhancements to the basic code please send me a note, or a copy of the modified source code, describing what you did, and I may include the change in future releases (with acknowledgment). If you just want to see a new feature added, also let me know, and I may add it.

Requirements

Minimal requirements

The programs have some minimal requirements for hardware and software. The requirements are:

- A Macintosh Plus or any model with equivalent 128K (or later) ROMs installed.
- System 4.2 or later (the programs may work with earlier system software but this has not been tested).
- Any version of HyperCard. However, some versions of HyperCard earlier than 1.2 may require that global variables be declared before they are used. Since the programs use a few global variables to store information between invocations you must declare these variables in a script, preferably in the openStack handler. See the section in this manual about the global variables for more details.

32-bit and A/UX

Use caution when operating on 32-bit clean environments such as A/UX since the programs have not been tested as being 32-bit clean. I have attempted to adhere to all of Apple's guidelines for writing 32-bit clean code, so there should be very few, if any, problems when used under such an environment. I can not, however, be sure that the programs will execute correctly.

Distribution

This section describes the terms under which these programs may be freely distributed.

Distribution

You may give out free copies of this software subject to the fairly unrestrictive conditions in the following section. The only thing I ask in return for your use of this software is that you send me a postcard, but this is only a request, not a condition for use.

Public license

Note: In this and the following sections titled "Copying Policies" and "No Warranty" the terms "this software" and "the software" refer to the source code, compiled object code, and manuals for the programs "ArrayList XFCN", "BinaryTree XFCN", "Research XFCN", "ErrorString XFCN", and "ErrorName XFCN".

I want to make sure that you have the right to give away copies of this software, that you receive source code or else can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things. To make sure that everyone has such rights, I have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of the software, you must give the recipients all the rights that you have. You must make sure that they receive or can get the source code. And you must tell them their rights (by including this notice).

Also, for my own protection, I must ensure that everyone finds out that there is no warranty for this software. Finally, if any of the software is modified by someone else and passed on, I want its recipients to know that what they have is not what I distributed, so that any problems introduced by others will not reflect on my reputation (each file contains a description of how you should note any changes made to the file).

Therefore I (Ari I. Halberstadt) make the following terms which say what you must do to be allowed to distribute or change the software.

Copying policies

1. You may copy and distribute verbatim copies of this software as you receive it, in any medium, provided that you conspicuously and appropriately publish on each file a valid copyright notice “Copyright © 1990 Ari I. Halberstadt”; keep intact the notices on all files that refer to this License Agreement; and give any other recipients of the software a copy of this License Agreement. You may charge a distribution fee for the physical act of transferring a copy.

2. You may modify your copy or copies of the software or any portion of it, and copy and distribute such modifications under the terms of Paragraph 1 above, provided that you also do the following:

   - cause the modified files to carry prominent notices stating who last changed such files and the date of any change (each source code file contains a header indicating how this is to be done); and

   - cause the whole of any work that you distribute or publish, that in whole or in part contains or is a derivative of the software or any part thereof, to be licensed at no charge to all third parties on terms identical to those contained in this License Agreement (except that you may choose to grant more extensive warranty protection to some or all third parties, at your option).

   - You may charge a distribution fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

   Mere aggregation of other unrelated software with this software (or its derivative) on a volume of storage or distribution medium does not bring the other software under the scope of these terms.

3. You may copy and distribute the software (or a portion or derivative of it, under Paragraph 2) in machine executable form under the terms of Paragraphs 1 and 2 above provided that you also do one of the following:

   - accompany it with the complete corresponding machine-readable source code, and manuals, which must be distributed under the terms of Paragraphs 1 and 2 above; or

   - accompany it with the information you received as to where the corresponding source code and manuals may be obtained.

For an executable file, complete source code means all the source code for all modules it contains; but, as a special exception, it need not include source code for modules which are standard libraries that accompany the operating system on which the executable file runs, or which are part of the development environment used to write the software.

4. You may not copy, sublicense, distribute or transfer the software except as expressly provided under this License Agreement. Any attempt otherwise to copy, sublicense, distribute or transfer the software is void and your rights to use the software under this License Agreement shall be automatically terminated. However, parties who have received computer software programs from you with this License Agreement will not have their licenses terminated so long as such parties remain in full compliance.

5. If you wish to incorporate parts of the software into other programs whose distribution conditions are different, please write to the author. This license agreement is only intended for general distribution, and I will often agree to different terms.

No warranty

Because this software is licensed free of charge, I provide absolutely no warranty. Except when otherwise stated in writing, Ari I. Halberstadt and/or other parties provide this software “as is” without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the program is with you. Should this software prove defective, you assume the cost of all necessary servicing, repair or correction.

In no event will Ari I. Halberstadt and/or any other party who may modify and redistribute this software as permitted above, be liable to you for damages, including any lost profits, lost monies, or other special, incidental or consequential damages arising out of the use or inability to use (including but not limited to loss of data or data being rendered inaccurate or losses sustained by third parties) the software, even if you have been advised of the possibility of such damages, or for any claim by any other party.

Appendix A. Bibliography

“HyperCard Script Language Guide: The HyperTalk Language” Addison-Wesley Publishing Company, 1988. Describes all the features and commands of HyperTalk, with a basic introduction to writing XCMDs and XFCNs. Don't expect to see "A real implementation of a spreadsheet better than Excel" here: this is primarily a reference book for people who want to use HyperCard to its fullest extent. Far better than any other book I have seen on the subject, this is the official documentation from Apple.

Henry McGilton and Rachel Morgan, “Introducing the UNIX System”, McGraw-Hill Book Company, 1983. An introductory text to using the UNIX operating system, this book was written with the end user in mind, and helped me write clearer descriptions of regular expressions. The book is somewhat out of date, which is its main drawback.

Kernighan, W. Brian and Ritchie, Dennis M., “The C Programming Language”, Prentice-Hall, 1978. This is the first edition of the standard text for programming in C. Though a somewhat terse and cryptic book, it was the only way to learn C for several years. Fortunately, there are now a plethora of other books on C to choose from. I would recommend purchasing the second edition of this text which has been updated for compatibility with the ANSI (*American National Standards Institute*) definition of the C programming language. This text is commonly referred to as “K&R”, in reference to the authors initials.

Sedgewick, Robert, “Algorithms”, 2nd ed., Addison-Wesley, 1988. A survey of the most important computer algorithms in use today. It has been used as the textbook for Dartmouth's introductory algorithms course, which is why I have this book. It is very useful for finding a good algorithm for some task or another, and as a starting point for locating better algorithms. Its main drawback is the use of cryptic names for many of the sample programs and its awful description of the network flow problem.

“XEROX Publishing Standards: A Manual of Style and Design”, Watson-Guptil Publications, 1988. The page layout and style used in these manuals is based on information from this book.

Zanny Whacko and Thumb A Ride, “The Hitchhikers Guide to Computers”, Ursa Minor Book Company, 2034. By far the most important text ever written about computers, its main attraction is the words “Don't Panic” which are inscribed on its cover. Certainly, its admonition regarding UNIX, “mostly harmless”, is the most useful insight ever provided on this popular yet archaic system.

Appendix B. About the author

The author of these programs may be reached at the following address:

Ari Halberstadt
11B Faxon St., #1
Newton, MA 02158

Tel. (617) 332-0290

If you have any comments about these programs he is eager to receive them. He also awaits a postcard from your home town.

The author enjoys programming the UNIX operating system, especially BSD4.x, Bourne Shell/C Shell and other standard UNIX amenities. Programming languages he uses include C, Objective C, Pascal, and MC68000 assembly. The Macintosh has consumed much of his time in the past year and a half; knowledge of application programming, interface design, and fancy XCMDs is firmly lodged in his synapses. Using MPW, including scripting, and THINK C, are Standard Operating Procedures.

The author's favorite unfinished program is a talking clock on the NeXT computer. His favorite dream computer is too far off to describe in less than 250 words. Racquetball and kayaking are probably more enjoyable than just programming. Dartmouth College is the best college he has attended.

Current computer interests include program specification, algorithm design, correctness proofs, applications of computers to analysis of biological systems, and hypothetical user interfaces. Robotics and certain fields in Artificial Intelligence seem like fascinating fields for future study. The author currently views his main purpose in life as the acquisition of knowledge and the understanding of the knowledge acquired.

Appendix C. Revision history

This section is to be used for recording any changes made to this manual. This is necessary since I do not want inconsistencies or mistakes introduced by others to reflect on my reputation, and, if the revisions improve this product, then the person who made the improvements should receive full credit. For consistency, please enter dates as Year-Month-Day.

Table 7. **Revision history**

Date	Name	Comments
90-07-18	Ari Halberstadt	This is an example entry

90-07-11 Ari Halberstadt Version 0.9