Chapter 10
Macintosh Interface

Abstract

This chapter describes the way that HForth interfaces to the Macintosh toolbox routines. Any ROM based Trap can be called. Macintosh records can be implemented using the HForth 'C' like structure tools. The *HForth Event Manager* handles all events which can be routed to applications that need them.

Introduction

The information in this chapter will not be needed for the typical HMSL program. You might find it useful, however, if you:

1) Need to use special Macintosh features.

2) Need to open a device driver.

3) Need to customize an application for Turnkeying.

HMSL is intended to be a host independant language. Programs written in HMSL should run on Macintoshes, Amigas, or any other machine that supports HMSL. This is so that HMSL can live long after Macintoshes are obsolete and everybody switches to some new computer. It also makes it possible to share code between machines. Thus the lowest level Macintosh calls have been hidden in libraries that look the same on the Mac and Amiga. This chapter describes how those libraries access the Macintosh specific tools. Warning - if you use anything from this chapter, it won't run on Amigas or other machines. Before using these tools, try to implement your application using the control grids and other host independant tools.

The bulk of the Macintosh Interface is contained in a few files. The General Mac related tools are in:

HSYS:Mac_Calls - defines TRAP: , PASS: and many calls,
HSYS:Mac_Structures
HSYS:Mac_Misc

If you need to call a Trap, check first in **HSYS:Mac_Calls** to see if it is already defined. If you study these chapters, you will see how the Macintosh interface works. The Macintosh specific tools that are music related are in the folder **HostDep**, known as **HH:**. Files like **HH:H4TH_GRAPH** and **HH:H4TH_EVENTS** are important files there. Also the MIDI and Clock related drivers are there.

If you want to use the Macintosh toolbox, you will need the *Inside Mac* documentation. Volumes 1,2 and 3 are critical. If you are real serious, you should probably get volumes 4, 5 and 6 as well.

Calling Macintosh ROM Traps

Most of the Macintosh operating system is contained in ROM and can be called using the **TRAP** mechanism of the 68000. **TRAPS** are a kind of exception, like an interrupt, that are identified by a hexadecimal number. For example, the procedure **LineTo()** is TRAP number $A891. To generate a TRAP call, HForth provides a word called **TRAP:** which compiles TRAP code. For example:

```
TRAP: A891
```

would compile a call to the LineTo() trap. You can find the TRAP number for any Mac ROM call by looking in Appendix G of Inside Mac Volume III.

Parameters to these TRAPS are passed on the return stack which is pointed to by A7. If there is a result, it is returned on the return stack. The integrity of the return stack is very critical to proper operation of the 68000 so be careful when using it. A small mistake will almost certainly crash the machine or generate an error. To make it easier to pass parameters, HForth provides a word called **PASS:** which takes parameters from the data stack and puts them on the return stack in the proper way.

Consider the procedure **SetCtlValue** which takes a 4 byte control handle and a 2 byte value (thus, 42 in the example below). In Pascal this would be described as:

```
SetCtlValue ( Control:handle, value:int )
```

To define a Forth word that corresponds to SetCtlValue, we would write:

```
: SetCtlValue()  ( Controlhandle value -- )
   PASS: 42  \ pass two parameters
   TRAP: A963
;
```

If we want to get a value back from a function, we need to make room for it on the return stack, then, pull it off after the call. Here is how we would define GetCtlValue which returns a 2 byte value. 2 byte values are sometimes called words so we have **W>R** , **WR>** and **0W>R** to help us here. Here is how it is defined in **HSYS:Mac_Calls**.

```
: GetCtlValue()  ( Controlhandle - value)
   0W>R  \ make room for result on return stack
   PASS: 4  \ Pass parameter. We could have said >R
   TRAP: A960
   WR>  \ get result from return stack
;
```

Notice that we put **()** at the end of these words. This is an HForth convention indicating that this word corresponds directly to a Toolbox call.

If you need to pass an OSTYPE, you can define then using:

OSTYPE: (string <name> --)

Defines a constant using the first four characters in the string. For example:

```
" CODE" OSTYPE: 'CODE'
```

Defines a 4 byte constant equal to $434F4445 which can be used as a resource type.

Using ToolBox Records

For some Toolbox calls, you will need to pass the address of a record. An example is:

```
PaintRect(bounds:rect)
```

This is defined in HForth as:

```
: PaintRect ( rect -- )
   >R TRAP: A8A2
;
```

We could have used PASS: 4 instead of >R. They are equivalent.

A record is a complex data structure that has many parts. A Rect record has 4 parts, or "members". They are top y, left x, bottom y, and right x. They must be contained inside the Rect data structure in the proper order. If we look in HSYS:Mac_Structures, we will see that RECT is defined as:

```
:STRUCT  RECT
   short rect_top
   short rect_left
   short rect_bottom
   short rect_right
;STRUCT
```

The **:STRUCT** word defines a record, or "structure", template that can be used to make many rectangle records. We declare the four members then terminate the record definition with a **;STRUCT**. These were all shorts but we could also have members called:

APTR (<name> -- , 4 byte absolute address pointer)

BYTE (<name> -- , 1 byte integer)

BYTES (n <name> -- , N bytes in this member)

LONG (<name> -- , 4 byte member)

SHORT (<name> -- , 2 byte member)

STRUCT (<structure_name> <name> -- , for nested structures)

To access these members, we can use the words **..@** and **..!** . These words will automatically take into account whether a member is 1, 2 or 4 bytes. Here is an example of declaring a RECT, then storing into and fetching from it.

```
RECT MYRECT
40 MYRECT ..! RECT_TOP
MYRECT ..@ RECT_TOP .  ( prints 40 )
```

.. (recaddr <name> -- memaddr , ofset address to named member)

..! (N recaddr <name> -- , store N into named member)

..@ (recaddr <name> -- n , fetch N from named member)

S! (N recaddr <name> -- , same as ..!)

In JForth, this does some extra address conversion not needed here.

S@ (recaddr <name> -- n , same as ..@)

In JForth, this does some extra address conversion not needed here.

HForth Event Handling

At the core of every Macintosh application is an Event Handler that gets events from the Macintosh operating system. In HMSL these events have to be sent to the HForth text window, the HMSL graphics window, the Text Editor, plus any other windows that users may open. To simplify this process, HForth has a technique for associating unique event handling functions with each window and menu.

All of the HForth event handling is done under **?TERMINAL**. As long as ?TERMINAL is getting called frequently, HMSL will respond to events. When the HForth kernel receives an event it checks to see what window or menu it is associated with. It then calls the apropriate event handler for that window or menu. Events are associated with a window using a **WindowTracker** structure. This structure is defined as:

```
:STRUCT WindowTracker
   aptr  wt_EVHandler  \ event handler
   aptr  wt_MDHandler  \ Mouse Down event handler
   aptr  wt_Window
;STRUCT
```

If the event is a MouseDown event, then FindWindow is called and the window part is stored in the variable WHICH-PART. A copy of the event is stored at the address LAST-EVENT. The wt_EVHandler can find out what the event type was using ..@, for example:

```
LAST-EVENT ..@ ER_WHAT
```

Here is the HForth definition of an event:

```
:STRUCT EventRecord
   short er_what
   long  er_message
   long  er_when
   long  er_where
   short er_modifiers
```

```
;STRUCT
```

Here are some Forth words related to this event handling system.

LINK.WINDOW<->TRACKER (window tracker --)

Link a window and an event tracking structure together.

EV.MENU.FUNC! (cfa index --)

Set the Menu handler for a menu with a Menu ID of INDEX. INDEX can be between 128 and 143. HMSL uses 128 to 137. You may use 138 to 143. The CFA must have the following stack diagram:

```
MenuHandler ( -- )
```

The menu handler can find out the item selected using:

```
WHICH-PART @ 15 AND  ( -- 1-15 , get item )
```

and then use CASE to process it.

WHICH-PART (-- var-addr , part of window hit in MouseDown)

WHICH-WINDOW (-- var-addr , window associated with last event)

WindowTemplate (<name> -- , define a new window template)

Window templates contain specifications for a window to be opened. This template can be initialized using WINDOW.DEFAULTS. Pass this to OpenWindow() to open a window based on the values in this structure. Here is the definition of WindowTemplate. See HSYS:DEMO_PAINT for an example of its use.

```
:STRUCT WindowTemplate
   long  wt_wStorage
   struct   rect  wt_rect
   long  wt_title
   short wt_visible
   short wt_procID
   long  wt_behind
   short wt_goAwayFlag
   long  wt_refcon
;STRUCT
```

This is defined in HSYS:MAC_STRUCTURES

Example Program that uses Toolbox

To see how these words are used in a real example, see the file **HSYS:DEMO_PAINT** which is a simple paint program with a menu. It opens a window, creates a menu, and lets the user draw lines using the mouse. It could be used as the starting point for a larger program.

Getting Rid of HMSL Menus for Turnkey

If you create a turnkey application, you may want to get rid of the HMSL Menus. You can make them disappear using this function:

```
DeleteMenu() ( id -- )
```

Pass the IDs for each menu you want to get rid of, then call:

```
DrawMenuBar()  ( -- )
```

Here are the IDs for the various HMSL Menus:

```
128 constant APPLE_MENU_ID
129 constant FILE_MENU_ID
130 constant EDIT_MENU_ID
131 constant INCLUDE_MENU_ID
132 constant FONTSIZE_MENU_ID
133 constant HMSL_MENU_ID
134 constant CUSTOM_MENU_ID
```

```
135 constant WINDOW_MENU_ID
```