Chapter 9
HForth Internals

Abstract

This chapter describes the internal workings of *HForth*, the Forth that supports HMSL on the Macintosh. It describes the Compiler, Optimizer, the structure of the dictionary, and differences between this and other Forths. This chapter is not for the beginning user, but rather for the HMSL programmer who wishes to understand the technical details of HForth.

HForth Compared to Other Forths

Forth has been well documented in a number of books and manuals (including this one, and to some extent, the HMSL Manual itself). If you know the differences between HForth and other Forths, then you can use other documents as a basis for using HForth. Many of you will be familiar with other Forths for the Macintosh (like *Mach2*, in which older versions of Macintosh HMSL were written), or *JForth* for the Amiga. In this chapter, I will point out similarities and differences between HForth and these two Forths. First the similarities.

HForth, Mach2 and JForth are all **32 bit** Forths. This means that items on the parameter stack, including addresses, are 32 bit numbers. This gives you greater precision then older 16 bit Forths, and a larger address space.

All three of these Forths are **subroutine threaded** with inline macros. Older Forths compile to a token language. These newer Forths compile directly to 68000 machine code. This makes them run about 2-3 times faster then a traditional Forth.

Now for some of the differences.

JForth caches the top of stack in D7 which is a 68000 data register. This gives it a speed advantage. **HForth and Mach2 do not cache.** This only matters if you are writing in assembly language. This is totally transparent to normal Forth code.

HForth has a contiguous dictionary. This means that the code dictionary is all in one piece. Most Forths are like this. Mach2 is somewhat unusual in that it segmented the Forth dictionary into 32K segments. The Macintosh prefers that code segments not exceed 32K. Unfortunately, however, this causes a lot of problems when writing large programs and is the main reason we no longer use Mach2 for HMSL. In Mach2 you cannot FORGET code from a previous segment. You also had to recompile all segments from scratch when making any segments. In HForth, you can FORGET very far back, almost into the kernel. You can also expand the size of the dictionaries. See the section on "Expanding the Dictionary".

Forth words have "headers" that contain the name of the word, a link pointer to the previous word, and other information. This is distinct from the executable code for the word. **HForth and Mach2 store their headers in an area separate from the code.** This makes it easier to get rid of the headers when creating a turnkey application. Stand-alone turnkey applications do not need the headers since they cannot compile Forth code. JForth gets rid of its headers by using Clone to build a completely separate image with only the necessary code and data. This is a better system but is more complicated and is not really needed by HMSL because HMSL is intended to be used as a Forth environment.

HForth and Mach2 use "absolute addressing" while JForth uses "relative addressing". In HForth and Mach2, fetch '@' and store '!', use actual 68000 addresses. JForth uses an "address" that is actually

an offset from the base of the JForth dictionary. This makes it easier to relocate the JForth dictionary in memory. It also, however, makes it more complicated to reference data outside the dictionary. In JForth you must convert addresses between relative and absolute when passing addresses to the operating system or when accessing hardware. In HForth and Mach2 you do not. You do, however, have to convert addresses to a relocatable form when saving them in the dictionary. In Mach2 this was complicated by having the dictionary broken up into segments.

In HForth we can simply convert the absolute address to a relative relocatable address by subtracting the base of the dictionary. When you are storing an address into the dictionary and expect it to be valid after you have done a SAVE-FORTH then you must do the following. **Use A! when storing addresses into the dictionary, and A@ when fetching them.** You only have to worry about this when they have to survive a SAVE-FORTH (and note that this is for very advanced users, the typical HMSL programmer doesn't need to worry about this). If you always initialize at run time then you don't have to worry about this. This is done internally by ODE, for example, when building method jump tables.

Summary Comparison of 3 Forths

	HForth	**JForth**	**Mach2**
32 Bit	yes	yes	yes
Subroutine Threaded	yes	yes	yes
Cache Top of Stack	no	yes	no
Separate Headers	yes	no	yes
Absolute Addressing	yes	no	yes
Contiguous Dictionary	yes	yes	no

Some Non-Standard Words

HForth is based on the *'83 standard* so you can use almost any textbook on Forth as a reference for HForth. There are a few words that are often in different in different Forths. Here is how they are defined in HForth.

' (<name> -- cfa , called "tick")

This word is IMMEDIATE and state sensitive in HForth. This means you can use it inside a colon definition just as you would outside a colon definition. In Forth '83 you would normally use ['] inside a colon definition. The old Forths, FIG and F79 did it the way HForth does it. Forth 83 has some good reasons to do it their way but it seems a little weird to change it now. Here is an example of how things would be done in HForth and in Forth '83.

HForth Example:

```
: FOO ." Hello" CR ;
: DOFOO  ( -- , execute FOO indirectly )
   '   FOO  EXECUTE
;
: SHOWCFA  ( -- , use "tick" in a definition )
   [COMPILE]   '   .
;
SHOWCFA  FOO   ( prints CFA of FOO )
' FOO .   ( prints CFA of FOO )
```

Forth '83 Example

```
: FOO ." Hello" CR ;
: DOFOO  ( -- , execute FOO indirectly )
   [']   FOO  EXECUTE     ( different !!!!! )
;
: SHOWCFA  ( -- , use "tick" in a definition )
   '   .     ( different !!!!! )
;
SHOWCFA  FOO   ( prints CFA of FOO )
' FOO .   ( prints CFA of FOO , same as HForth )
```

" (<string>" -- $address , create string)

This word is used for string literals. It returns the address of the count byte of a string. Some Forths return the address of the first character and the count. This is silly because you can easily get that from the $address using COUNT but it is difficult to go the other way. It is also easier to manipulate the string using just one parameter, the $address.

HForth Example

```
" Hello"   COUNT   TYPE  ( print string )
```

F83 Example

```
" Hello"  TYPE   ( easier here but harder for other stuff )
```

DO (limit start -- , mark the beginning of a loop)

This is used with LOOP to create a looping construct, for example:

```
: COUNTUP   ( N --  )
   0 DO  I  .  CR LOOP
;
10 COUNTUP
```

This will print the numbers from 0 to 9. It does not reach the limit value. 0-9 is ten numbers, however, which makes sense. In HForth, if you pass a LIMIT that equals the START then it won't EXECUTE.

```
2  COUNTUP  ( prints   0  1 )
1 COUNTUP  ( prints 0 )
0 COUNTUP  ( prints nothing )
```

This is what you would expect. Most Forths will always execute the LOOP at least once. Thus:

```
0 COUNTUP  ( prints 0  , in some other Forths )
```

." (<string>" -- , print string to output)

In Forth '83, it is illegal to use ." outside of a colon definition but HForth lets you use it anywhere.

Macintosh Head/Tail Optimization

The Macintosh Forth optimizes the code between words to avoid excessive stack manipulation. This results in a 10-30% speedup. The optimizer is on by default. You can turn it off or on using **OPTIMIZER.OFF** or **OPTIMIZER.ON**.

HForth Memory Map

There are three main memory areas in HForth. (See figure 9.1.) The **Code** area contains the executable 68000 code, variables, data structures, object data, etc. The top of the code area is returned by **HERE**. The defined code grows up toward high memory. The unused memory above the last defined word is shared with the data stack. The bottom of the data stack is at the top of the Code area. The data stack

grows down. If you compile too much code into the dictionary, the code will collide with the data stack. See SAVE-FORTH in the glossary for information on how to expand the dictionary.

The **Application Data** area is pointed to by the 680x0 address register A5. It contains variables used by the kernel and buffers like the TIB.

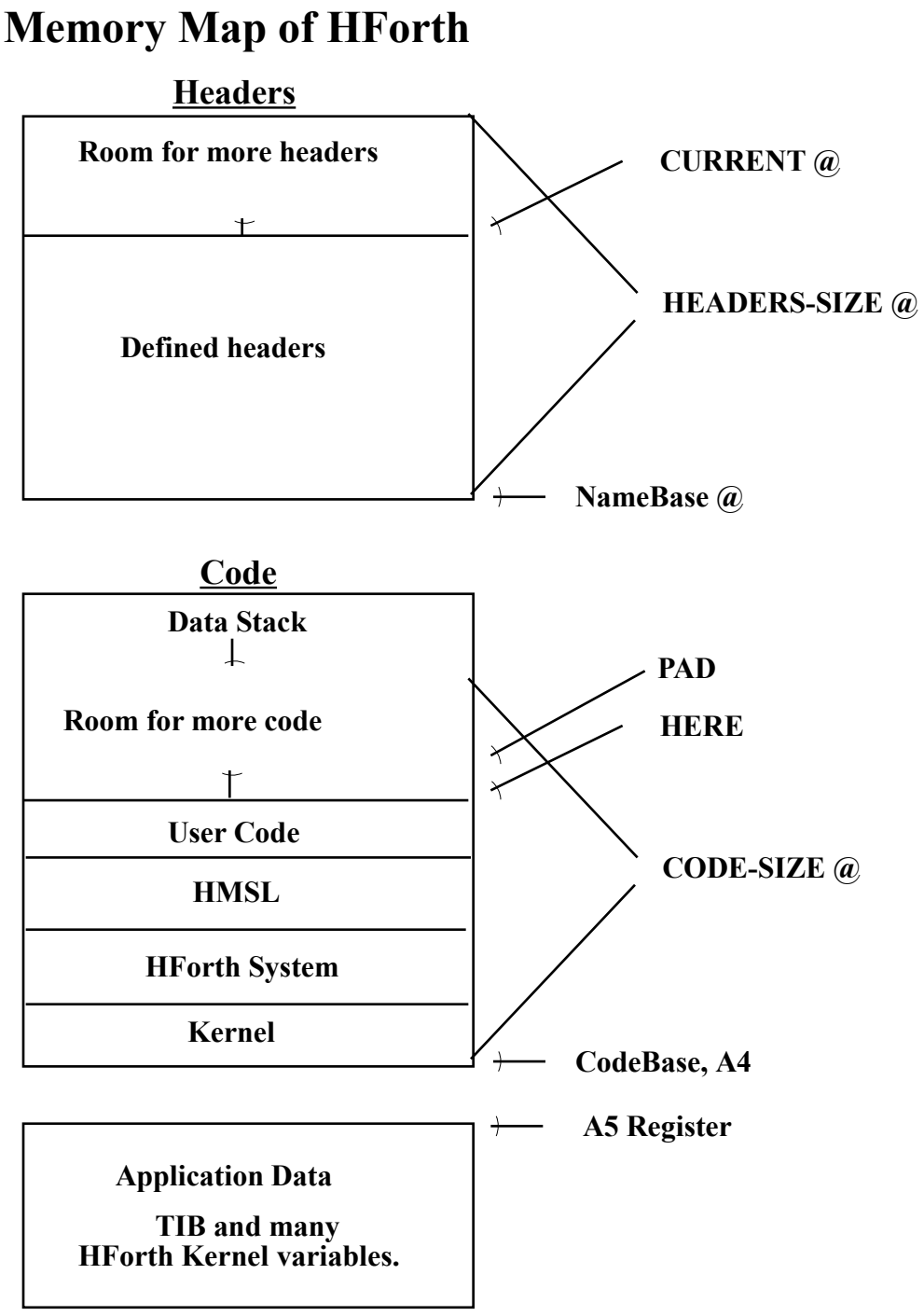

Figure 9.1

The **Headers** areas contains a linked list of headers. Each header contains the name of a Forth word, a *link field* which points to the previously defined word's name, a pointer to the associated code, and a *flag field*. The Link Field pointer is stored as a relative offset from NameBase. This makes it relocatable. Even though the absolute address may change depending on where HForth gets loaded into memory, the relative positions will not change. The pointer to the code for the name is stored as an offset from *CodeBase* for the same reason. The variable **CURRENT** contains the address of the last word defined. (See figure 9.2.)

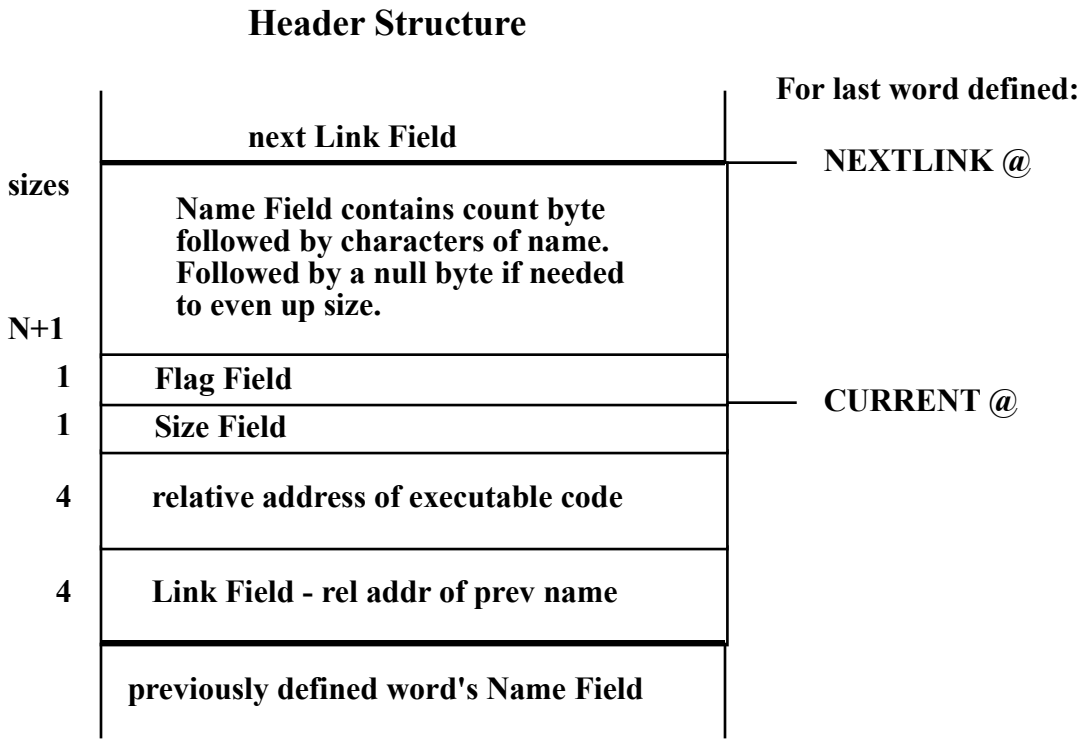

Figure 9.2

Name Field = contains word name and count byte

Flag Field = contains TAIL, HEAD, INLINE and IMMEDIATE flags

Size Field = size of code in bytes, not counting RTS

Link Field = points to previously defined word

Code Field = contains executable 68000 code

To better understand this dictionary structure, let's try some experiments. Let's define some words then get the address of the name of the most recently defined word.

```
HERE .  ( Address where code for FOO will go. Remember it. )
: FOO  ( -- ) ." Hello!" CR ;
: COUNTUP ( -- ) 10 0 DO i . LOOP ;
CURRENT @ ID.  ( should print COUNTUP )
```

COUNTUP is the most recent word defined. CURRENT, therefore, will return the address of its Name Field. ID. takes an NFA and prints the name. We printed the value of HERE right before defining FOO. The code for FOO should go into the Code area at that address. To check this enter:

```
' FOO .  ( should be same as HERE was )
```

Let's try to reproduce by hand what happened in the previous line. First let's scan the dictionary until we find FOO. We start with **CONTEXT**, which is a variable like CURRENT, that tells us where to start looking.

```
CONTEXT @  ( get top word )
DUP ID.  ( print COUNTUP )
10 - ( get link field for COUNTUP, could also use N>LINK )
@ NameBase +  ( link back to previous word, convert to absolute )
DUP ID.  ( This should be FOO. We found it! )
```

We now have the NFA of FOO. To find out where the code is for FOO we could use NAME>, or do the equivalent by hand which is to subtract 6 from the NFA, fetch the value there and convert it to absolute.

```
6 -
@ CodeBase +
DUP .
' FOO .  ( this should be the same as the previous line )
```

We now have the Code Field Address for FOO. If we want to execute it we can enter:

```
EXECUTE  ( should see "Hello!" )
```

This will give use some idea of how the compiler looks up words in the dictionary and executes them.

Here some dictionary related words you can look up in the HForth glossary:

```
' >NAME CODE-SIZE HEADERS-SIZE N>LINK NAME> PREVNAME SAVE-FORTH
```

Explanation of Flag Field

This field contains single bit flags that control how the compiler treats the word. The user will not normally need to use this information. Here is an explanation of each flag:

```
Bit#  Mask  SetBy       Meaning
  0     1   IMMEDIATE   Words is executed at compiled time.
  1     2   SMUDGE      Words is hidden from FIND.
  2     4   SET.HEAD    Word is HEAD optimizable.
  3     8   SET.TAIL    Word is TAIL optimizable.
  4   $10   SET.INLINE  Word is compiled as inline macro, no JSR.
```

Here is an example of using SET.INLINE to make an INLINE code macro. This technique is useful for making very fast macros or when the return stack must be accessed in a way that would be disrupted by a subroutine call. This word will add the number on the return stack to the top number on the data stack. Without using SET.INLINE, our word R+ would add the return address from the subroutine call instead.

```
: R+  ( n -- n+r )   r@ + ; SET.INLINE
: TEST
   23 >R      ( push 23 on return stack )
   100 R+  .  ( add it to 100 )
   5555 R+  . ( add it to 5555 )
   RDROP      ( remove 23 from return stack )
;
```