Chapter 2

How to Learn HMSL

Introduction: Paths to learn the system

Learning HMSL can be a bit like like learning a foreign language. It would be nice to know *every word* in the language, but you can *get by* after learning just the most important ones. Many composers have written interesting pieces in HMSL after just a few days of learning. Those same composers are still learning new techniques after using HMSL for several years.

At the foundation of HMSL is the **Forth** language. A working knowledge of Forth is necessary to learn HMSL. You may already know Forth but if not we recommend using the tutorials in the Forth documentation. These can be found in the JForth manual for the Amiga, or in the Macintosh Supplement. There is a self-quiz at the end of this chapter to test your Forth knowledge. If you can pass this quiz then you know enough Forth to continue learning HMSL.

HMSL has many different aspects to it. You can learn the parts that most interest first, then learn other parts later. The best way to learn is to have fun and play. We'll describe in this chapter several different possible paths you can take to learn HMSL. Pick the one that most interests you and learn that one first.

Path 1 - MIDI, low level Scheduling and Timing, System Exclusive.

Path 2 - HMSL Hierarchies, Object Oriented Programming, Shapes, Players, Instruments, Jobs, Collections, Actions.

Path 3 - User Interfaces, Object Oriented Programming, Graphics, Building Interactive Screens.

Path 4 - Tools: Score Entry System, Sequencer.

Path 5 - Advanced HMSL Hierarchies, Interpreters, Markov Chains

Path 6 - Amiga Local Sound, Samples and Waveforms, Tuning

Here is a list of what to read in the HMSL manual for each path. You may want to check things off as you learn them.

Path 1 - MIDI

___ Tutorial 1 in Chapter 13 on MIDI . Learn how to send MIDI messages to synthesizers, Note On, Control, etc.

___ Examples in Chapter 13 on the MIDI Parser. Learn how to respond to MIDI messages from a keyboard or other input device

(for more advanced ideas):

___ Chapter 14 on Timing and Scheduling. Learn how to schedule MIDI events in the future using the hardware clock.

___ Chapter 13 on MIDI System Exclusive messages.

___ Chapter 13 on MIDI Files

Path 2 - HMSL Hierarchies

___ First ODE Tutorials Chapter 4

___ Overview of Morphs

___ Shape Tutorials in Chapter 5

___ Player Tutorials in Chapter 6

___ Instrument Tutorials in Chapter 7

___ Collection Tutorials in Chapter 8

___ Jobs Tutorials in Chapter 10

___ Perform Tutorials in Chapter 12

Path 3 - User Interfaces

___ ODE Tutorial in Chapter 4

___ Tutorials on Interactive Controls in Chapter 15

___ Tutorial on Graphics in JForth Manual or Macintosh Supplement

Path 4 - Tools

___ Chapter 16 on Score Entry System

___ First Tutorial on the Sequencer in Chapter 17

Path 5 - Advanced HMSL Hierarchies, do Path 2 first.

___ Interpreters Tutorial in Instruments Chapter 7

___ Structures and Markov Chains, Chapter 9

___ Translators and Translator Functions, Chapter 11

Path 6 - Amiga Local Sound and Tuning

___ Chapter on Amiga Local Sound in Amiga Supplement

___ Chapter on Amiga Instruments in Amiga Supplement

___ Section on Tunings in Translator Chapter

___ Chapter on Timing and Scheduling

Other Tips on Learning HMSL

Disk-Based Examples

Study the examples on disk. There are many example files on disk that you can compile and run. We recommend reading the file first by using a text editor so you know what it is supposed to do. You may want to make a copy of these files and change them to experiment with different ideas. These files will be in the pieces folder/directory on the HMSL_User disk.

These files contain *many useful code examples*, but they also contain a great deal of important information about *programming style*, which is very important, especially to the beginning user and

programmer. Good programming habits can save you an enormous amount of time, energy, and frustration, and make your composing and performing more enjoyable. Pay careful attention to the uses of indentation, comments, upper and lower cases, naming of routines, use of special techniques like ANEW, TERM and INIT words, IF.FORGOTTEN. Learning these techniques early on will really help your music programming.

HMSL Source Code

Look at the source code for different parts of HMSL. If you want to know how something works, look at how it was written. You can use the word **FILE?** to find out what file something was defined in. For example, enter:

```
FILE?  CHOOSE
```

This will tell you the name of the file that CHOOSE was defined in. Most of the HMSL *source code files* can be found in the Source folder/directory on the HMSL_Source disk. A method for a Class of objects may be defined in its Superclass' file. To find out where to look for the definition of a method, use **METHODS.OF**. For example:

```
METHODS.OF  OB.PLAYER
```

This will list all the methods defined for a given class. Notice that some of the methods are from Jobs, Collections, and other classes.

Before You Start, A Forth Quiz!

Before tackling HMSL, you should already have a basic understanding of Forth. Here is a short quiz to test your Forth chops:

1) *[Use of variables]* Define two variables called VAR-1 and VAR-2. Write a Forth word that implements the following algebraic statement:

```
VAR-2 = (VAR-1 * 3) + 5
```

2) *[Text output, simple control structures]* Write a word in Forth to say "Hello" repeatedly until you hit a key. Each "Hello" should be on its own line.

3) *[Stack manipulation]* Complete the following definition:

```
: STACK.MANIP ( a b -- a a b b )
   ?????????
;
```

4) *[Editing and compiling a file]* Use the editor to write a program that adds up the first N odd numbers.

5) *[Object oriented programming]* Create an OB.ARRAY object with ten items. Fill it with various numbers. Write a word to add them up. Print the array.

6) *[Extra Credit]* What did the Zen master say to the hot dog vendor?

If you want to check your answers, they can be found at the end of this chapter.

Syntax Used in Code and Manual

Whenever possible, we conform to certain mnemonically motivated *syntactical conventions*. We *highly recommend* that users follow these to increase comprehensibility. We have found, through many years of teaching HMSL, that if you get into the habit of following these simple conventions at the beginning, your code will be far more readable, and easier to debug. Some of these conventions are as follows:

- In general, in example code, parentheses indicate comments. All code in this manual is generally in upper case, but —

- In general it is a nice idea to use upper case in your own code mainly for the names of words, and for selected control words like DO, LOOP, BEGIN, UNTIL, CASE, ect. Use lower-case for the code in the definition. For example:

```
: MY-WORD
   some.other.word
;
```

  Note that this is not the way the code appears in this manual, but look at the examples on the disk, or the source code for HMSL itself for a good guide.

- A hyphen in a word indicates a variable, the name of instance of some class, or a "noun" of some sort, as in these examples:

```
FOO-BAR ( variable called FOO-BAR )
COLL-SEQ-2 ( SEQUENTIAL COLLECTION number 2 )
ACT-5 ( ACTION number 5 )
```

- An underscore in a word indicates a constant, often a filename, or some "noun" that is not a variable. Here is an example:

```
DA_NUM_CHANNELS  ( #channels on Amiga )
```

- A "." (period) in a word indicates some sort of "verb," or function. Periods can also be used in METHODS (see below, and the Chapter on Object Development Environment). For example,

```
TIME.INIT
```

initializes the real-time-clock.

- A ":" (colon) at the end of a word indicates that the word is probably a METHOD, as in this example:

```
ADD: ( a method used to add data to an object )
```

- A "$" (dollar sign) in a word name or stack diagram indicates that the word involves Forth strings that are represented by an address of a count byte followed by the characters of the string.

```
$.  ( $string -- , print out a string )
for example:    "  Hello!" $.
```

- A parameter in angle brackets indicates a string which FOLLOWS the word. This parameter is *not* passed on the stack as a number.

```
FOPEN ( <filename> -- fileid | 0 )
for example:    FOPEN  77MHD:DATAFILE     or
'C  ( <forthname> -- cfa )
for example:    'C MIDI.NOTEON  .HEX
```

- Some simple file-name prefixes are used:

```
ajf_ ( Amiga JForth only )
mmac_ ( Macintosh Mach 2 only )
obj_ ( related to object oriented code )
```

- Words from a single software module often have a common prefix to distinguish them, as in these examples:

```
MIDI.NOTEON ( turn on a MIDI note )
MIDI.PRESSURE ( set MIDI channel pressure )
DA.PERIOD! ( set period for Digital Audio )
```

- When PUT(...): is used in a method name, that implies that an argument is needed on the stack, as in this example:

```
ACT-5 PUT.ACTION: ACTION-TABLE
( Takes the address of ACT-5 and places it in the action table )
```

- When USE...: is used in a method name, that means that no argument in general is needed, as in this example:

```
USE.DURATIONAL: PLAYER-1
( Tells PLAYER-1 to start using durational scheduling, and no argument
is needed ).
```

- Forth stack diagrams follow the standard syntax: a double hyphen (--) indicates the stack. The contents of the stack before the word is executed are to the left of the hyphens, the contents of the stack after the word is executed are to the right of the hyphens. Top of the stack is always furthest to the right! In other words, the value closest to the hyphen on input is the top of the stack, the value furthest from the hyphen on output is the top of the stack. A comma after the stack output begins a comment on the word; " | " indicates "or", such as "1 | 0 ."

Answers for Forth Quiz

Answer to 1)

```
VARIABLE   VAR-1
VARIABLE   VAR-2
:  DOMATH  ( -- , set var2=var1*3+5)
   VAR-1 @
   3  *   5   +
   VAR-2  !
;
( now test it )
20  VAR-1  !
DOMATH
VAR-2    ?     ( should say  65 )
```

Answer to 2)

```
:  SAYHI   ( -- say hello almost forever )
   BEGIN
      ."  Hello!"   CR
      ?TERMINAL ( wait for key hit from terminal )
   UNTIL
;
SAYHI
```

Answer to 3)

```
:  STACK.MANIP   (  a  b  --  a  a  b  b  )
   OVER SWAP  (  -- a  a  b )
   DUP
;
\ or
:  STACK.MANIP   (  a  b  --  a  a  b  b  )
   SWAP  DUP    ( -- b  a  a )
   ROT  DUP
;
2  5  STACK.MANIP   .S
```

Answer to 4)

```
:  ODD.SUM   ( N  -- SUM , sum first N odd numbers )
   0   ( initial sum )  SWAP       0
   DO I  2*  1+  ( -- odd )  +
   LOOP
;
3   ODD.SUM   .
7   ODD.SUM   .
3   3   *   .
7   7   *   .    ( What???    Isn't math fun!?!?!)
```

Answer to 5)

```
OB.ARRAY   MY-ARRAY     ( instantiate an array )
10   NEW: MY-ARRAY      ( make room for 10 numbers )
23   2   TO: MY-ARRAY   ( put in various numbers )
93   4   TO: MY-ARRAY
721  5  TO: MY-ARRAY
PRINT: MY-ARRAY   ( print out array )
: ADDEMUP     ( -- SUM  , add up numbers in MY-ARRAY )
   0  ( initial sum )
   SIZE: MY-ARRAY  0
   DO ( -- sum )
      I   AT: MY-ARRAY   +
   LOOP
;
ADDEMUP  .  CR   ( test it )
FREE: MY-ARRAY ( deallocate the memory allocated by NEW: )
```

Answer to 6)

"Make me one with everything." (thanks to Mark Trayle)