Appendix B
Programming Tips

Morph Memory Allocation

Don’t forget to do a NEW: before you using morphs that require it. Remember, there is no default NEW: for morphs. Make sure to FREE: your morphs when you're finished with them. If you don't deallocate memory that you have allocated, then eventually the Amiga (and to a lesser extent, the Macintosh) can run completely out of RAM. Also, remember that in defining a new class, the method NEW: calls FREE: itself.

Local Variables in a Morph’s Functions

As we have pointed out several times in the manual, it’s a good idea to use local variables as a habit in the function for a morph, when that function is expecting something on the stack. For example, for a job function that either stops a job or lengthens its duration depending on a random number:

```
: MY-JOB.FUNCTION { job -- , }
   50 CHOOSE DUP
   5 < ( -- val flag )
   JOB GET.DURATION: [] ( -- val flag dur )
   500 > OR ( -- val flag )
   IF
      DROP JOB SET.DONE: []
   ELSE
      JOB GET.DURATION: []
      + JOB PUT.DURATION: []
   THEN
;
```

If you almost always do this with these types of functions —for example, an interpreter should always have the stack diagram

{ element shape ins -- }

you will find it a lot easier to write and maintain code, and won’t have to worry nearly so much about stack manipulation.

ANEW and INCLUDE?

The word ANEW is important to use at the beginning of each file, and will make your development cycle a lot easier, efficient, and more productive. You use it in the form:

ANEW TASK-MY_FILE

where MY_FILE is the name of the file. The first time you compile the file, ANEW will simply create a dummy word in the dictionary called TASK-MY_FILE, or whatever name you give. If you now recompile the file, ANEW will recognize the word TASK-MY_FILE and FORGET all the words in the dictionary down to that word. You can now recompile the rest of the file without getting duplicate definitions for the words in the file. If you have a piece that involves several files and you have ANEW at the beginning of each file then you can create a loader file that contains conditional INCLUDEs for each file, for example:

INCLUDE? TASK-ZIP_TOOLS ZIP_TOOLS

INCLUDE? TASK-ZIP_CLASSES ZIP_CLASSES

INCLUDE? TASK- ZIP_STUFF ZIP_STUFF

INCLUDE? TASK-ZIP_MAIN ZIP_MAIN

Now imagine that you compile the whole piece and then realize that there is a bug in the file ZIP_CLASSES. You can bring up the file ZIP_CLASES and edit it. Then recompile ZIP_CLASSES. The ANEW will automatically FORGET the code down to the beginning of ZIP_CLASSES. Keep editing and compiling until ZIP_CLASSES works. Then include your loader file. The first two files will already be loaded so the load file will skip them and then load the rest of the piece, ZIP_STUFF and ZIP_MAIN. This is a relly handy way to work when you have large pieces. You will notice that HMSL files all haveANEW at their beginning.

You can actually call the word following ANEW anything you want (FOO, GLORBYGRIBUGLICK, MICHAEL_JACKSON, etc.), but for clarity sake and by convention we suggest using the full name of the file preceded by TASK- , and reserving TASK- for that purpose only. See the chapter on Forth additions to HMSL for more details on the use of ANEW .

Important: On the Macintosh, the **FILE?** utility uses the **TASK-xxx** word to display what file a word was defined in. If you do not follow the convention of TASK-filename, then FILE? will not be very useful on the Macintosh.This is also described in the HForth supplement.

Cleaning Up Afterwards

With things like players and instruments, where there are lots of possibilities for changing settings after the default INIT: , you should probably get in the habit of changing them back after you're finished using them, for example, when you finish one piece and are going to do another. The values you put in these morphs by altering them will of course remain and could adversely effect the next piece. For this purpose, you can also use the DEFAULT: and DEFAULT.HIERARCHY: methods.

Make sure you CLEAR: the ACTION-TABLE when you forget actions. In general, always FREE: morphs when you forget their children, so that they do not reference morphs that do not exist. This is a *very* common bug in the development cycle of pieces that use the Action-Table.

You may want to define a file called something like MY_UTILITIES. In this file, you can put small Forth or HMSL routines that you use often, and then you can load this file with all your pieces. One very handy word to define in that file would be a word like:

```
:  CLEANUP ( -- )
   CLEAR: SHAPE-HOLDER
   CLEAR: ACTION-TABLE MP.RESET
;
```

Note that this word cleans up the Shape-Holder, the Action-Table, and the MIDI Parser. The HMSL main pull-down menu contains an entry called RESET. This performs a general system reset, including: clearing the Active Object list, deallocating MIDI and Amiga Instruments, redrawing the current screen, and a few other housekeeping functions. If used with care, it can be helpful for clearing out the system. You can also enter HMSL.RESET from the keyboard. If you hang up HMSL, and the Shape-Editor or the Action-Table is still displayed, you can type HMSL.CLOSE, which will close the window that HMSL opens for these screens.

Incremental Testing

Get in the habit of making simple interactive tests of various user-defined functions like: instruments (try just calling the NOTE.ON: and NOTE.OFF: methods with the proper arguments, OPEN: first!); actions (make sure you turn them on with ACT.ON: or they won't do anything); productions (just execute them, and see if they do what they're supposed to); and behaviors (remember to pass it the structure). This can save you a lot of wondering about why the system isn't doing what you think it's supposed to, and is one of the main advantages of Forth!

When testing, it often helps to PRINT: the various objects you are working with. You will often notice the cause of errors this way. PRINT.HIERARCHY: is another useful word that can quickly tell you if your morphs are being created properly.

Recursive Morph Posting

A morph may not directly or indirectly execute itself. This means that a collection (or structure) should not contain itself, nor should any of that collections children or grandchildren contain the parent collection. If this occurs, HMSL will get stuck looping in the tree, or worse. It is possible for many different morphs to call the same morph as long as they are not descendants of each other. If this occurs, the morph called will send a DONE: message to its parent and start execution under the new parent. If we represent a hierarchy as an outline, we can show an illegal and a legal form.

This hierarchy is **ILLEGAL**:

```
COLL-SEQ-1
   COLL-SEQ-2
      PLAYER-1
      COLL-SEQ-1 ( ILLEGAL NESTING !!!)
   COLL-SEQ-3
```

This is a **LEGAL** hierarchy:

```
COLL-PAR-1
   COLL-SEQ-1
      PLAYER-1
      COLL-SEQ-3
   COLL-SEQ-2
      COLL-SEQ-3
      PLAYER-2
```

When the COLL-SEQ-3 is played by COLL-SEQ-1 it will turn off the other one if it is still playing but will not crash. If the top collection were sequential then there would be no interference or problem of any kind.

Action Screen Text

To prevent long names from overflowing the boxes provided in the Action Screen, use short names.

Speed

Once a piece of code has been debugged, it is possible to turn off some of the error checking to speed up ODE. If you set the variable OB-IF-CHECK-BIND to false, then there will be no error checking on Late Binding. If you want to speed up a morph's data access, you can enter:

```
FALSE DO.RANGE: MY-MORPH
```

This will disable range checking on array indices for that morph. If you set the variable IF-RANGE-CHECK to false then newly defined morphs won't have range checking. There are two words, RUN.FASTER and RUN.SAFER that set these variables thus allowing you to easily make trade offs between error checking and speed.

One tip for the user regarding execution speed is that REPEAT-COUNT based operations are, in general, faster than operations that traverse up the hierarchy. That is, if you want to have a player repeat 10 times, and that player is part of a collection, use a repeat count in the player if possible, not in the collection. Of course this won't always be appropriate to your musical ideas, and even if you occasionally hear lags in very complex hierarchies HMSL should never get out of "sync". It just occasionally needs to "catch up to itself". As we optimize some of the very low level code, this catching up will become less and less noticeable, and for most situations it should not be a problem even at present.

Mac Object Addresses

Remember that when you pass an object around on the Macintosh, you are not really passing the address but a relocatable token. This token can be converted to an address using REL->USE. You should probably never have to do this but if you try to dump an object using its address, you will need to know this.

Who's Bug Is This?

Note that since HMSL is a complete programming environment, it is both possible and likely that user-written code can crash the system. For example, a user-written response for an action might be simply to overwrite RAM. The programming emphasis has not been on crash-proofing the system. However, every reasonable attempt has been made to enable the user to make musical experiments without crashing things, and we assume responsibility for, and will fix, any bugs within HMSL.