Appendix A
Morphs

Introduction

This appendix provides an overview of some methods and ideas that apply to all morphs, as well some more technical and theoretical information on the basics of the HMSL morph, short for morphology. This material may be of interest to the advanced user, a user interested in learning some general shortcuts and techniques, or to those wishing for a more theoretical discussion of the class from which the usual HMSL data structures (collections, players, etc.) are defined.

Most Important Information

Morphs are the **primary data structures** in HMSL. Shapes, collections, players, structures, etc. are all morphs. Several **stock morphs** of each type have been defined for ease of use. Many pieces in HMSL involve building up a **hierarchy** of morphs and then executing that hierarchy. This is usually done using the word **HMSL.PLAY**. Thus to execute a collection called MY-COLLECTION, you would enter:

```
MY-COLLECTION  HMSL.PLAY
```

To execute a morph from within a running piece, use START: , eg.

```
START: COLL-P-1
```

Use **PRINT.HIERARCHY:** to print out the complete hierarchy. This is useful for debugging. When you are done using a hierarchy, you should use **DEFAULT.HIERARCHY:** then **FREE.HIERARCHY:** to put things back to their original condition and to free allocated memory.

Morph-lists are described here as well. The morph-list called **SHAPE-HOLDER** contains a list of the shapes that are available for editing in the Shape Editor. You can ADD: new ones and should **DELETE:** them or **CLEAR:** the SHAPE-HOLDER when you are done.

Introduction to Morphs

(Note on terms: "Morphs" is short for "morphologies," or ordered sets of data, the units of form. Thus, for example, morphogenesis is the growth and evolution of morphs and morphological metrics are distance functions between morphs.)

From the standpoint of the HMSL object-oriented programming environment, **OB.MORPH** is the class that all of the other morphs are derived from. It defines the basic methods for *message-passing* and *multitasking*. The definitions described here apply to all of the other morphs. All of the HMSL morphs (collections, shapes, structures, actions, productions, jobs, players) are subclasses of OB.MORPH, and inherit its methods, some of the most important of which are NEW: , ADD: , PUT: , EXECUTE: , STOP: , and FREE: . OB.MORPH itself is a *subclass* of OB.ELMNTS.

FREE: and FREE.HIERARCHY: are quite important, and are defined for each morph. FREE: frees the space for a given morph, FREE.HIERARCHY: frees the space of a given morph and all of its children (component morphs).

Several morph methods, including PUT: , GET: , TO: , AT: , MANY: , ED.AT: , and ED.TO: , DIMENSION: , NEW: , FREE: , RESET: , NEXT: , and MANYLEFT: are basic to all subsequent objects, and are used frequently in the source code for defining higher level morph class methods. The user can and should take advantage of the inheritance structure of HMSL's object-oriented environment.

In many cases in this manual, morph methods are redefined for each subclass of morph even though the method is not redefined significantly from the superclass' definition. These methods, like FREE: , etc., are crucial, and their redundant documentation is used for purposes of convenience and reminder for the user.

Executing a Morph Hierarchy

A hierarchy of morphs is executed by HMSL using a special message system that communicates up and down the "morph tree." Understanding how this works will help you understand how to nest morphs effectively.

A morph is executed by sending it an EXECUTE: message. This is done by words like HMSL.PLAY which will execute a morph and enter HMSL. When a collection or a structure is executed it in turn sends EXECUTE: messages to its component morphs. These component morphs in turn execute their component morphs and so on. The order and manner in which the component morphs are executed depends on the behavior of the collection. The morph that calls a component morph is called the **INVOKER**. The **EXECUTE:** method always returns immediately even though it may take several minutes for the hierarchy itself to finish playing.

Since a morph cannot predict when its component morph(s) will finish executing, it simply waits for them to send a message back saying they are *done*. When an invoking morph executes a component morph, the invoker passes the component morph the time it should start, and the invoker's own address as a "return address." When the component morph finishes, it sends the time it finished and its own address back to the invoking morph using that "return address." The invoking morph can then either invoke more morphs or, if it is done, send a **DONE:** message to its invoker. In this manner, every morph in the tree can eventually be executed.

Players and *jobs* have no component morphs. Instead of invoking component morphs, they instead *post* themselves to the multitasker. Players play elements of their shape when sent **TASK:** messages from the multitasker. When a player or job is done, it sends a DONE: message to its invoker as before. The time sent back is the time it started execution plus the sum of the durations of all the elements it played.

Executing versus Posting

The words execute and post are often used in confusing ways so we shall attempt to clarify their meaning here. The basic way to execute a morph is this:

```
time invoker EXECUTE: morph
```

The parent, or "invoker," passes its address to the morph to be executed. Thus the executed morph "knows" who to tell when it is done. When the morph is at the top of a hierarchy, there is no invoker so you should pass a zero for the invoker parameter. For example,

```
TIME@ 0 EXECUTE: MY-COLLECTION
```

would allow you to start the execution of MY-COLLECTION yourself from inside of HMSL, as if from an action. This assumes that the HMSL multitasker, the Polymorphous Executive, is running. In this case, current time is used. There is a shortcut method celled **START:** for starting a morph at the top of a hierarchy. Instead of the above example, you could have used:

```
START:  MY-COLLECTION
```

To start execution of a hierarchy when the Polymorphous Executive is not already running, you could use HMSL.PLAY as follows:

```
MY-COLLECTION HMSL.PLAY    ( executes morph and starts HMSL running )
```

HMSL.PLAY is a simple utility that allows you to specify a morph, and execute it as well as HMSL itself.

Execute and Post are defined as follows:

Execute 1. To instruct a morph to perform its "duty" (which depends on the type of morph it is). A piece can be played by executing the top morph in the hierarchy using the HMSL word EXECUTE: or START: 2. To execute the Forth word whose code field address (CFA) is on the stack. This is done using the Forth word EXECUTE (notice lack of colon, this is not a method!).

Post To place a morph in the Active Object List, ACTOBJ, using the word AO.POST . This results in a morph acting in parallel with other morphs by being sent TASK: messages in rapid succession. When a player or job is executed, it posts itself to the multitasker. Players and jobs are the only morphs that currently support being posted.

Some Basic Morph Methods

The following methods are defined for OB.MORPH and are inherited by the other morphs. OB.MORPH is a subclass of OB.ELMNTS .

Method	Stack diagram
ABORT:	(-- , aborts a morph and the morphs that invoked it)
DEFAULT:	(-- , sets a morph to its state at INIT:)
DEFAULT.HIERARCHY:	(-- , does DEFAULT: down a hierarchy)
DONE:	(end-time sender -- , notify invoker)
EXECUTE:	(start-time invoker -- , Execute a morph)
FREE.HIERARCHY:	(-- , sends FREE: message to everything in hierarchy)
GET.DATA:	(-- N , get user data)
GET.INVOKER:	(-- invoker , return morph that executed this one)
PUT.DATA:	(N -- , put user data value in morph)
PRINT.HIERARCHY:	(-- , prints "children" of a given morph)
TASK:	(-- , perform a single time slice)
START:	(-- , start executing a morph)
STOP:	(--)
?HIERARCHICAL:	(-- flag , true if can contain other morphs)

OB.MORPH Methods

ABORT: (-- , aborts a morph and the morphs that invoked it)

This is used by HMSL.ABORT to clean up the system when a piece is interrupted by hitting the HMSL close box. This avoids the problem of having instruments left open, MIDI notes left ON, etc.

DEFAULT: (-- , sets a morph to its state at INIT:)

Sets all values of a morph to what they were when the morph was INIT:ed, like REPEAT-COUNTS, weights, etc., but does not FREE: or NEW: or alter data set by these words. In general, resets values of instance-variables to their INIT: values.

DEFAULT.HIERARCHY: (-- , does DEFAULT: down a hierarchy)

DEFAULT:'s a morph and all its children, and all their children (etc.).

DONE: (end-time sender -- , notify invoker)

This method receives the "done" message from the executed morph. DONE: is also responsible for executing the next child in a sequential morph. DONE: is primarily used by the morphs themselves and the HMSL multi-tasker (Polymorphous Executive), and is usually invisible to the user. However, it can be accessed by the user in the creation of new morph classes. For example, the user might want to redefine the DONE: message for a new morph class to BEEP every time it sends a "done" message to its invoker.

EXECUTE: (start-time invoker -- , Execute a morph)

Causes a morph to execute itself. When that morph is then done it will send a DONE: message back to its invoker. Essentially, EXECUTE: is the message that travels "down" the hierarchy, and DONE: is the message that travels "up".

Note that actions do not use this method, but instead are TASK:ed by the Action Table.See the chapter on PERFORM for more information.

FREE.HIERARCHY: (--)

FREE:'s all components of a morph (all its children, and all their children, and so on). That is, it FREE:'s the entire hierarchy downward from the morph specified. This is an extremely useful and important method for

morphs, for by using it, the user does not have to keep track of the particulars of a hierarchy in order to FREE: all its space.

GET.DATA: (-- n , get user data)

This is handy for fetching some user defined value associated with a morph. Typically used by production and job functions.

GET.INVOKER: (-- invoker , return morph that executed this one)

This can be used inside an executing morph to find out the parent morph.

INIT: (--)

Initializes morph. Does INIT: as for the superclass (OB.ELMNTS) and clears any reference to an invoking morph.

INIT: is very specifically defined for each morph class, and the reader can consult the source code for more details. Normally, this is a transparent routine, done when a morph is defined, and the user should not have to call this method. INIT: calls DEFAULT: which should be used in new morph classes for setting default values. The main difference between DEFAULT: and INIT: is that DEFAULT: can be called by itself without messing up the allocated memory pointers set by NEW: .

PRINT.HIERARCHY: (-- , prints "children" of a given morph)

Prints indented list of morph's tree of component morphs. For morphs that don't contain other morphs, PRINT: just prints their name. For players, it prints the component shapes, and for productions, the names of the routines whose CFAs are stored in the production.

PUT.DATA: (n -- , put user data value in morph)

This is handy for storing some user defined value associated with a morph. Typically used by production and job functions.

REGISTER: (time -- , set start time for morph)

This is called when a tasked morph (players, jobs) is executed. Internal.

START: (-- , start executing a morph)

Performs a TIME@ 0 EXECUTE: . This is used when executing a morph that is at the top of a hierarchy. HMSL must be running to hear anything.

STOP: (--)

Terminates execution of a morph, and all its children. STOP: can be executed at any time to "turn off" a morph. In fact, whenever a morph is executed, it first checks to see if it is already being executed, and STOP:'s itself first if it is.

TASK: (-- , perform a single time slice)

This is used by the multitasker to cause a morph to do whatever it needs to do at that instant. Whatever it does it must be fast; no delays or waits can occur. The morph may decide to do nothing at that time. If the morph is completely done then it will unpost itself.

?HIERARCHICAL: (-- flag)

True if morph necessarily contains other morph addresses, false if not. TRUE for collections, structures, tstructures, players. FALSE for jobs, actions, productions, and shapes.

This method is used internally by methods like PRINT.HIERARCHY: , DEFAULT.HIERARCHY: and FREE.HIERARCHY: , but may be useful for the user in writing routines that need to "know" the "type" of morph that they are acting upon.

Morph Lists

Morph lists may be used to keep track of instances of morphs that have been created by the user. They comprise lists of such morphs, which may be added to by the user when new morphs are declared.

The following morph tracking lists exist, all of type OB.OBJLIST:

```
SHAPE-HOLDER
COLL-HOLDER
STRUCT-HOLDER
PLAYER-HOLDER
PRODUCTION-HOLDER
JOB-HOLDER
```

At present, only SHAPE-HOLDER is immediately useful, since shapes put in it will be available in the Shape Editor.

It is important to clear these holders when you are through (if you have used them). Otherwise problems can occur if you FORGET a morph that is listed in a holder. The holder would contain a reference to something that no longer exists. This problem can be avoided by using stock morphs which are never forgotten. A similar problem can also occur if you have a morph, for example, MY-COLL, reference another morph, for example, MY-PLAYER, and then FORGET MY-PLAYER without clearing MY-COLL.

An OB.OBJLIST is a subclass of OB.LIST, which is a subclass of OB.ELMNTS. It is essentially a one dimensional OB.ELMNTS class that is designed to contain a list of other objects.

Morph List Methods

The following methods are defined for morph lists.

Method	Stack diagram
ADD:	(morph -- , add morph to holder)
CLEAR:	(-- , clears holder)
DELETE:	(morph -- , delete a morph from list)
EXTEND:	(n -- , lengthens morphlist by n)

Morph List Methods

ADD: (morph -- , add morph to holder)

ADD: is a method for the class OB.OBJLIST.

CLEAR: (-- , clears holder)

CLEAR: is a method for the class OB.OBJLIST.

Example:

```
CLEAR: SHAPE-HOLDER .
```

You may clear a holder without first forgetting its component morphs, but not vice versa. You should not forget a morph and leave it "hanging" in the holder!

DELETE: (morph -- , removes one morph from list)

If you just want to remove one morph, use this instead of CLEAR: .

EXTEND: (n -- , lengthens morphlist by n)

Used for creating bigger holders if needed.

OB.OBJLIST-Related Forth Words

Word	Stack diagram
ML.CLEAR	(-- , clear record of defined morphs.)
ML.FREE	(-- , frees the things listed in the holders)
ML.INIT	(--)
ML.PRINT	(-- , print contents of all holders)
ML.TERM	(-- , frees the holders)

OB.OBJLIST-Related Forth Words

(Note that since these words act on all the morphlists, they are not methods but Forth words).

ML.CLEAR (-- , clear record of defined morphs.)

When a user places a morph in a holder, that holder can be used to keep track of that class of MORPH. ML.CLEAR clears all these holders. It does not FORGET the actual morphs themselves.

If you FORGET your morphs you should call ML.CLEAR.

ML.FREE (-- , frees the things listed in the holders)

Does FREE: on the shapes, collections, productions, structures, and players listed in the four holders. Works by sending messages to all the morphs listed instructing them to deallocate their space, then clearing the holders.

ML.INIT (--)

Initializes the holders, by doing a NEW: command, and sets a variable called ml-if-init to TRUE. This is done internally by HMSL.INIT .

The storage made available by this command is as follows:

SHAPE-HOLDER 64

COLL-HOLDER 64

STRUCT-HOLDER 32

PLAYER-HOLDER 32

PRODUCTION-HOLDER 64

JOB-HOLDER 64

ML.PRINT (-- , print contents of all holders)

Prints the shapes, collections, structures, productions, and players.

This is like listing a directory of files on a disk, except it lists all MORPHS added since ML.CLEAR was called.

ML.TERM (-- , frees the holders)

Does an ML.FREE to deallocate all the shapes, collections, productions, and structures listed in the holders, then deallocates the holders themselves.

Stock Morphs

Some stock shapes, jobs, players, collections, structures, tstructures, instruments, and actions are defined by the system for convenience. They are just declared and initialized; no storage is allocated unless you do it with NEW:. The morphs are as follows:

```
SHAPE-1 through SHAPE-8
PLAYER-1 through PLAYER-8
COLL-S-1 through COLL-S-4 (sequential collections)
COLL-P-1 through COLL-P-4 (parallel collections)
JOB-1 through JOB-4
ACT-1 through ACT-16
INS-MIDI-1 through INS-MIDI-4 (simple MIDI instruments, see Instruments
chapter)
INS-AMIGA-1 through INS-AMIGA-4 (simple local sound instruments, see
Instruments chapter)
```