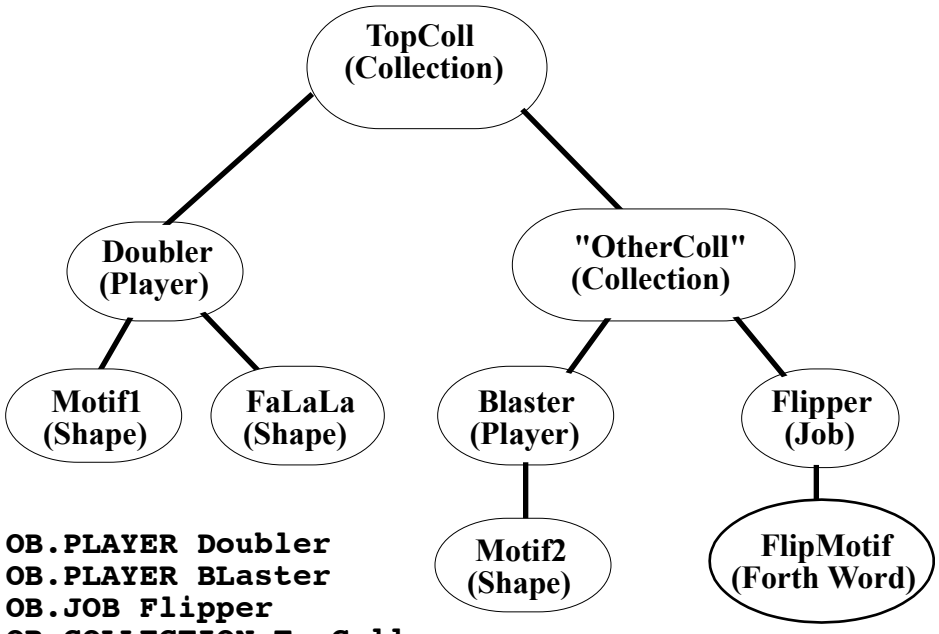
Chapter 8
Collections

Most Important Information

Collections are used to hold and *execute other morphs*. They are the primary building block of a hierarchy. Collections can execute their morphs either sequentially, in parallel, or in a customized order using a *behavior*. You can put things in a collection using }**STUFF:** or **NEW:** and **ADD:** . You can also set a repeat count using **PUT.REPEAT:** . You can specify custom functions to execute when a collection is started, repeated, or stopped. Collections can be executed using **START:** if the HMSL scheduler is running. For example, to execute a collection called COLL-S-2, enter:

```
HMSL.START
START: COLL-S-2
```




```
OB.PLAYER Doubler
OB.PLAYER BLaster
OB.JOB Flipper
OB.COLLECTION TopColl
OB.COLLECTION OtherColl
STUFF{ Doubler OtherColl }STUFF: TopColl
STUFF{ Motif1  FaLaLa    }STUFF: Doubler
STUFF{ Blaster Flipper   }STUFF: OtherColl
STUFF{ Motif2 }STUFF: Blaster
STUFF{ 'C FlipMotif }STUFF: Flipper
```

Example HMSL Hierarchy

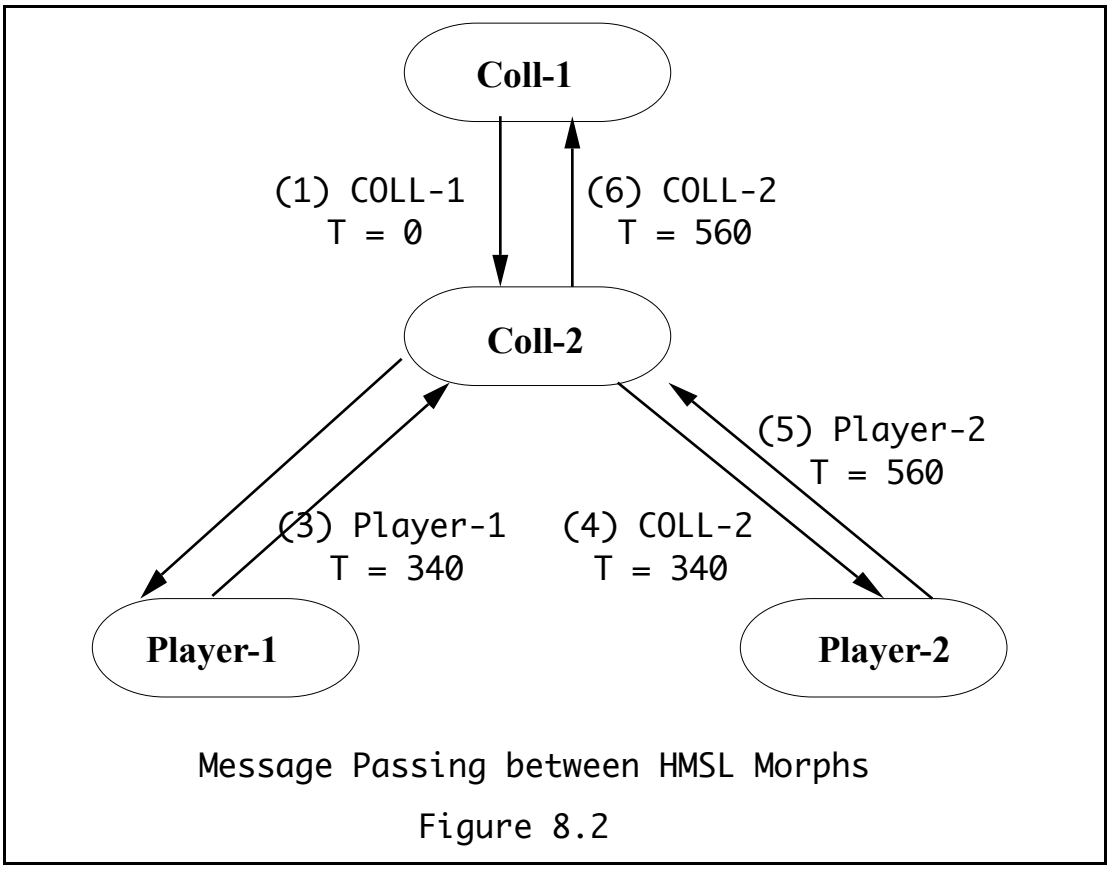
Figure 8.1

COLLECTIONS: Brief Introduction

Collections contain other morphs, usually players, jobs, and other collections. The fact that collections can contain other collections allows for complex tree structures, or *hierarchies*. Morphs contained within a collection are often referred to as its *children*. The shapes in players and the functions in jobs are typically the lowest level nodes, or leaves, of the hierarchy tree. Figure 8.1 shows a typical HMSL hierarchy along with some of the code used to construct such a hierarchy.

Collection Execution

When a collection is executed, it typically executes its children *sequentially* or in *parallel*. Morphs use a *message passing* scheme to coordinate this execution. When a collection executes sequentially it sends a starting time and its own address to its first child. When that child finishes, it sends a DONE return message to its parent consisting of the time it finished and its address. The parent collection then executes the next in line, and so on. When a *parallel collection* executes, it sends messages to all of its children at the same time. It then counts DONE return messages. When the last message is received, it either repeats the process or passes a DONE message to its parent. By setting a flag, these messages can be echoed to the screen to assist in debugging complex pieces.



Message Passing between HMSL Morphs

Figure 8.2

The number of repetitions is controlled by setting a *repeat count.* Collections, players, jobs, and Structures all have a repeat count that can be set by the composer. These morphs also have START, REPEAT, and STOP functions. Whenever one of these morphs is started, for example, it executes its START function. This allows user written functions to be easily embedded in the hierarchy. These functions could be used to randomize shape data, trigger other morphs, reconfigure a hierarchy, place new functions in the MIDI Parser, etc.

Collection Customization

Collections can be further customized. Instead of being strictly sequential or parallel, a *custom behavior* can be specified that determines the order in which the children are played. A behavior is passed the *address of the collection* and returns the *indices of the children* to play next and a *count*.

```
YOURBEHAVIOR ( collection -- i1 i2 ... in n )
```

For example, a collection could have a random behavior, or decide which child to play next based on user input, etc. The following is an example of a user written random behavior.

```
: RAND.BEHAV  ( coll -- i 1 , play 1 randomly )
    MANY: []  ( how many are there ? )
    CHOOSE  1
;
```

Tutorial 1: A Simple Hierarchy

Required: MIDI synthesizer(s) set to respond to channels 1 & 2.

Let's suppose you want to play two melodies in parallel. We have seen in the chapter on players how to play a single melody. Collections allow us to combine, or "collect", one or more morphs together. We can then play those morphs one after the other, *sequentially*, or together, in *parallel*.

Enter this tutorial into a file so you can experiment with it. Enter in a file:

```
ANEW TASK-COLTUT
\ Instantiate the morphs we will need.
OB.PLAYER  PL1
OB.PLAYER  PL2
OB.COLLECTION PCOL1
\
: PC.INIT  ( -- , setup players and collection )
   PREFAB: PL1   ( quick and dirty setup )
   PREFAB: PL2
   STUFF{  PL1  PL2  }STUFF: PCOL1
   50 PUT.REPEAT:  PCOL1
;
: PC.TERM  ( -- , clean everything up )
   CLEANUP: PCOL1
;
IF.FORGOTTEN  PC.TERM
```

In **PC.INIT**, the **PREFAB:** methods dynamically instantiate a shape and a MIDI instrument for the player. The shape is filled with a melody based on a random walk.

The **STUFF{** command takes whatever was between the brackets and stuffs it into PCOL1. This is an easy way to build a hierarchy. The next line tells COL1 to repeat 50 times when played.

The **IF.FORGOTTEN** command will cause the collection to be cleaned up automatically if we forget the code that defined it. This technique handy if you are compiling a file over and over again and don't want to have to call PC.TERM by hand each time. Most HMSL programmers get in the habit of using IF.FORGOTTEN often.

By default, a collection is *parallel.* You have defined a parallel collection above, even though it is not specified.

To hear this collection, compile the file and enter directly:

```
PC.INIT
PRINT: PCOL1
PCOL1  HMSL.PLAY   ( Click on close box when done )
```

You should have heard two melodies playing in parallel. Now let's change the way the collection behaves by telling it to play its two players sequentially instead of in parallel. Let's also set the repeat counts of the players to 2 to make this clearer. Enter directly:

```
2 PUT.REPEAT:  PL1
2 PUT.REPEAT:  PL2
ACT.SEQUENTIAL:  PCOL1
PCOL1 HMSL.PLAY
```

Using the method **ACT.SEQUENTIAL:** will transform this collection into a *sequential* collection. To set PCOL1 back to its original state, enter:

```
ACT.PARALLEL:  PCOL1
```

Tutorial 2: Nested Collections

Collections can contain players, jobs, structures *and* other collections. This allows you to build very large and complex hierarchies. Let's build a sequential collection that contains a player and a parallel collection. We can use the parallel collection from the previous tutorial. In this example we will build a melody by hand instead of using **PREFAB:** . Go back into the same file and remove the line that sets the repeat count of PCOL1 to 50, then add the following at the end of the file:

```
OB.SHAPE  SH3
OB.MIDI.INSTRUMENT  INS3
OB.PLAYER  PL3
OB.COLLECTION SCOL1
: SC.INIT  ( -- , set up sequential collection )
   32 3 NEW: SH3
   STUFF{    \ build shape by hand
      12  5  70
      12  6  80
      24  7  90
      24  8  90
       6  14 70
       6  14 70
   }STUFF:  SH3
\ Tell PL3 to use SH3 and INS3
   SH3  INS3  BUILD: PL3
\ Use MIDI preset 7 in instrument.
   7 PUT.PRESET:  INS3
   STUFF{  PL3  PCOL1  }STUFF:  SCOL1
   20 PUT.REPEAT:  SCOL1
   ACT.SEQUENTIAL: SCOL1
   PC.INIT  ( init parallel collection )
   PRINT.HIERARCHY:  SCOL1
;
: SC.TERM  ( -- )
   CLEANUP:  SCOL1
   PC.TERM   ( redundant in this case )
;
IF.FORGOTTEN SC.TERM   \ for automatic cleanup if forgotten
```

Compile the file and enter directly:

```
SC.INIT
SCOL1  HMSL.PLAY
SC.TERM
```

You will hear the melody from PL3 alone, followed by PL1 and PL2 in parallel.

See the file **HP:DEMO_COLLECTION** for another example.

Collections: Technical Description, Behaviors, Methods

Technical Description

As described above, *collections* are executable morphs that contain other morphs, unlike shapes which are "inactive bags of arbitrary data". Most *executable morphs* may be put into a collection. Shapes and actions may not. Collections can contain other collections, players, jobs, productions, or structures.

Collections typically execute their component morphs either *sequentially* or in *parallel*. In *sequential mode*, a collection will wait for the first morph to finish before executing the next one. It considers itself finished when the last of its morphs sends its DONE: message back. In *parallel mode*, it will execute all of its component morphs simultaneously. It considers itself finished when all of its morphs have sent their DONE: message back. Either of these two modes can be selected using the ACT.SEQUENTIAL: or ACT.PARALLEL: methods.

Note that any combination of these two may be obtained by putting one inside the other; that is, a parallel collection can contain a sequential collection, or vice versa. By arranging parallel and

sequential collections in a hierarchical manner, very complex patterns can be achieved. In fact, having these two types of collections is necessary and sufficient for the specification of all possible "morph trees" (with the obvious exception of a "recursive tree", when a given morph is called by itself at some depth of the tree).

OB.COLLECTION is a subclass of OB.MORPH, so it inherits the methods and instance variables of OB.MORPH.

Collections have an internal repeat count, which is used by the **EXECUTE:** method to determine how many times to do the collection when it is EXECUTE:ed. There are two methods, **GET.REPEAT:** and **PUT.REPEAT:** that can be used with this.

Putting a *0 in the repeat count of a morph "disables" that morph*. If the collection is executing while the repeat count is changed to 0, then it will stop when it finishes the current iteration. To stop a morph immediately you can use the **STOP:** method. This is useful for turning off morphs from actions (as a response to a stimuli), behaviors, productions, etc. If a morph is running through several repetitions and you want it to stop at the end of the current repetition, use the **FINISH:** method.

Behaviors

In addition to being either sequential or parallel, a collection can use a custom function called a *behavior* to determine the order of execution of its morphs. When a collection is executed, it calls its behavior to determine the *index* of the next component morph, and EXECUTE:'s it. At some time later, when those morphs are finished, they will send a DONE: message back to the collection . The collection will then select the next set of morphs using the behavior as before. The collection will continue until its behavior returns a count of 0, defined by convention as a terminator. The REPEAT-COUNT will then be decremented, the collection will then be reset, and the above process repeated until the REPEAT-COUNT runs out.

Behavior Definitions: There is a strict syntax for user-defined behaviors, which has the stack diagram:

```
MY.BEHAV  ( coll -- i1 i2 ... in N )
```

where i1 i2 ... in are indices of morphs in that collection, and N is the number of indices. If the count, N, is zero, don't play any, we're done.

Every collection has a nodal weight, which may be used inside collections, or more specifically, by a collection's behavior, in the determination of order for that collection.

An Example Behavior

`BH.RANDOM ( coll -- index 1 | 0 )`

BH.RANDOM is a simple behavior currently defined by HMSL. It returns either the *index of one of the component morphs plus a 1*, or it returns *0 to stop*. The 0 is as likely as getting any particular morph. Here's its definition:

```
: BH.RANDOM  ( coll -- index 1 | 0 )
   MANY: [] ( get length of collection )
   1+ CHOOSE DUP  ( select a random index+1 or 0 )
   IF 1- 1  ( -- index 1 )
   THEN
;
```

The preceding definition is a bit "slick," but it will be educational for the user to figure out how it works.

As in other such HMSL functions, it can be useful to make COLL a local variable in a behavior definition. For example, here's an alternative definition to the behavior above, which also randomly changes its own repeat count, and prints it's name and new repeat count (note how many times we have to refer to the local variable):

```
: BH.RANDOM  { coll -- index 1 | 0 }
   COLL MANY: [] ( get length of collection )
```

```
            1+ CHOOSE DUP  ( select a random index+1 or 0 )
            IF
               1- 1  ( -- index 1 )
            THEN
            5 CHOOSE COLL PUT.REPEAT: []
            COLL NAME: []
            ." New repeat count is: "
            COLL GET.REPEAT: [] . CR
    ;
```

Collection Methods

Method	Stack diagram
}STUFF:	(0 morph-1 morph-2 morph-3 ... morph-n -- ,) (does NEW: and adds morphs to collection)
ACT.PARALLEL	(-- , execute collection in parallel)
ACT.SEQUENTIAL	(-- , execute collection sequentially)
ADD:	(addr-morph -- , add to collection)
BEHAVE:	(-- index , executes a collection's behavior)
EXECUTE:	(time invoker -- , execute all component morphs)
EXTEND:	(n -- , enlarges collection to be able to) (include n more morphs)
FINISH:	(-- , stop after current repetition)
GET.REPEAT:	(-- repeat-count)
GET.WEIGHT:	(-- weight , get nodal weight)
INIT:	(--)
NEW:	(max-morphs -- , allocate room for morphs)
PRINT:	(-- , print it)
PUT:	(addr-morph index -- ,) (put in appropriate place in collection)
PUT.BEHAVIOR:	(CFA-behavior -- , sets behavior to use)
PUT.REPEAT:	(repeat-count --)
PUT.WEIGHT:	(weight --)
RESET:	(--)
STOP:	(-- , immediately stops the coll. & its children)
START:	(-- , immediately starts the coll. & its children)

Table 8-1. Collection Methods

?EXECUTE: (time invoker -- time true | false)

If the morph is done immediately, it will return the virtual time it finished and a TRUE. If the morph takes time to play, like a player or job, it will post itself and return FALSE. When it is finished it will send a DONE: message back to the parent invoker.

ACT.SEQUENTIAL: (-- , behave sequentially)

ACT.PARALLEL: (-- , behave in parallel)

ADD: (addr-morph -- , add to collection)

Example: To add PLAYER-1 to a collection called COLL-S-1, say:

```
PLAYER-1 ADD: COLL-S-1
```

BEHAVE: (-- index , executes a collection's behavior)

This is used internally by the collection to get the index of the next component morph which will be executed.

This is also useful for testing to see if a behavior is working as expected.

EXECUTE: (time invoker -- , execute all component morphs)

This is primarily for internal use but can be used externally for testing. It is called by HMSL.PLAY to start the execution of a hierarchy. Motion down the hierarchy is achieved when a morph uses EXECUTE: on its component morphs. EXECUTE: can be very useful in seeing if an action or a production is working correctly.

To execute a collection from within an action or a job, you should use START:

EXTEND: (n -- , enlarges collection to be able to include n more morphs)

GET.REPEAT: (-- repeat-count)

See PUT.REPEAT: .

GET.REPEAT.DELAY: (-- delay , fetch delay)

GET.REPEAT.FUNCTION: (-- cfa , function to exec at repeat)

This method can be used to execute a user function when a collection or other morph, repeats. In many cases Productions are no longer needed. The stack diagram of the word must be:

```
MYFUNC  ( morph -- , whatever )
```

See HP:DEMO_REPFUNC for an example.

GET.REPETITION: (-- count , fetch which repetition)

GET.START.DELAY: (-- delay , fetch execution delay)

GET.START.FUNCTION: (-- cfa , function to exec at start)

GET.STOP.DELAY: (-- delay , fetch delay)

GET.STOP.FUNCTION: (-- cfa , function to exec at stop)

GET.WEIGHT: (-- weight , get nodal weight)

Returns weight of the collection (0 by default, or value set with PUT.WEIGHT:) on the stack.

NEW: (max-morphs -- , allocate room for morphs)

The number of dimensions need not be specified since it is always one.

Example:

```
OB.COLLECTION MY-PARALLEL
2 NEW: MY-PARALLEL
```

PRINT: (-- , print it)

Prints the values of the collection, including everything printed for morphs plus the nodal weight of the collection and repeat count.

PUT: (morph index -- , put in appropriate place in collection)

Example:

```
PLAYER-1 2 PUT: COLL-S-1
```

puts PLAYER-1 in the third position of a sequential collection. Assumes MANY is 3 or greater. See SET.MANY: in the OB.ELMNTS documentation.

PUT.BEHAVIOR: (CFA-behavior | 0 -- , set behavior of collection)

Store the CFA of an executable routine that implements the behavior of the collection. You can choose a predefined behavior or define your own. Behaviors are passed the address of the collection and return the indices of the next morphs to execute, and a count. Behaviors may also do anything else the user wishes, such as alter the weights of various morphs, rearrange other morphs, change shapes, read a stimulus, play a note, etc. They can be any executable code that has the right stack diagram (the address of the collection is expected on the stack).

Behaviors should decide when to end by returning a 0 for the count of morphs.

Example:

```
'C BH.RANDOM PUT.BEHAVIOR: MY-COLLECTION
```

If you call PUT.BEHAVIOR: with a zero, the collection will go back to being either parallel or sequential.

PUT.REPEAT: (repeat-count --)

Set the number of times to repeat a collection when executed.

PUT.REPEAT.DELAY: (delay -- , set delay between repeats)

PUT.REPEAT.FUNCTION: (cfa -- , function to exec at repeat)

PUT.START.DELAY: (delay -- , store execution delay)

Delay the start of a collection when it is executed. This can be used to stagger players in a parallel collection for phasing effects by giving each player a different delay time. See HP:POLYPHASE for an example.

PUT.START.FUNCTION: (cfa -- , function to exec at start)

PUT.STOP.DELAY: (-- delay , store delay between repeats)

PUT.STOP.FUNCTION: (cfa -- , function to exec at stop)

PUT.WEIGHT: (weight --)

The nodal weight is intended for use by some behaviors to decide which morph to execute next in a collection. Note that GET.WEIGHT: and PUT.WEIGHT: may be usefully called by the programmer from productions and actions as well.

SET.DONE: (-- , set done flag to terminate)

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: (--)

STOP:'s execution of a collection and all its children.

}STUFF: (stuff{ morph-1 morph-2 ... morph-n -- , does NEW: then adds morphs)

}STUFF: (pronounced "bracket stuff") is an extremely basic and useful method (defined for *all morphs*) for building a hierarchy. It is a substitute for NEW: and ADD: . For example, to build a collection with 5 players in it:

```
OB.COLLECTION MY-COLLECTION
5 NEW: MY-COLLECTION
PLAYER-1 ADD: MY-COLLECTION
( repeat this line with new players ... )
```

```
PLAYER-5 ADD: MY-COLLECTION
```

or the following syntax would work:

```
OB.COLLECTION MY-COLLECTION
STUFF{ PLAYER-1 PLAYER-2 PLAYER-3 PLAYER-4 PLAYER-5
}STUFF: MY-COLLECTION
```

This can be used for collections, players, structures, productions, actions, and jobs, but in the latter three, note that 'C must be used before each "addition" to the morph (since you are adding CFA's).

Philosophy behind Collections

Collections are the *parent class* for most HMSL morphs, and thus productions, actions, jobs, players, and even structures share nodal weights (in actions they are called *priorities*). Although the original design of HMSL more or less intended collections to be a lower level morph than structures, the current generality of the design makes this no longer a necessary assumption. However, since structures are somewhat "intelligent" regarding their execution sequence (with the capability of having a transition matrix associated with them, as well as a behavior), it may be useful to think of the following data hierarchy, from lowest to highest: shapes, players, collections, structures.

A cognitive model of collections might be that they are any arbitrary hierarchical grouping of data whose grouping is morphological. By hierarchical we mean "morphs can contain each other", and by morphological, we mean ordered, or that "one morph can be said to come after another". Although similar to the kind of statistical temporal gestalts suggested by, for example, Tenney in *Meta + Hodos,* collections are by definition "ordered." In HMSL, that ordering may be user-defined (type it in), or algorithmically computed (productions, jobs, actions or even collections themselves may alter the morphology of a collection). One of the intentions in the HMSL-provided utilities is that the user be able to experiment with deliberate reorderings of collections, by means of various compositional algorithms, distance functions, stimuli, and the like.