Chapter 6
Players

Most Important Information

Players schedule the interpretation and performance of shape data. In other words, they play the shapes that they contain. Some of the most important player methods are: **PUT.INSTRUMENT:** , **PUT.DUR.DIM:** , **PUT.DUR.FUNCTION:** , **PUT.ON.DIM:** , **PUT.REPEAT:** , **PUT.REPEAT.FUNCTION:** , and **START:**

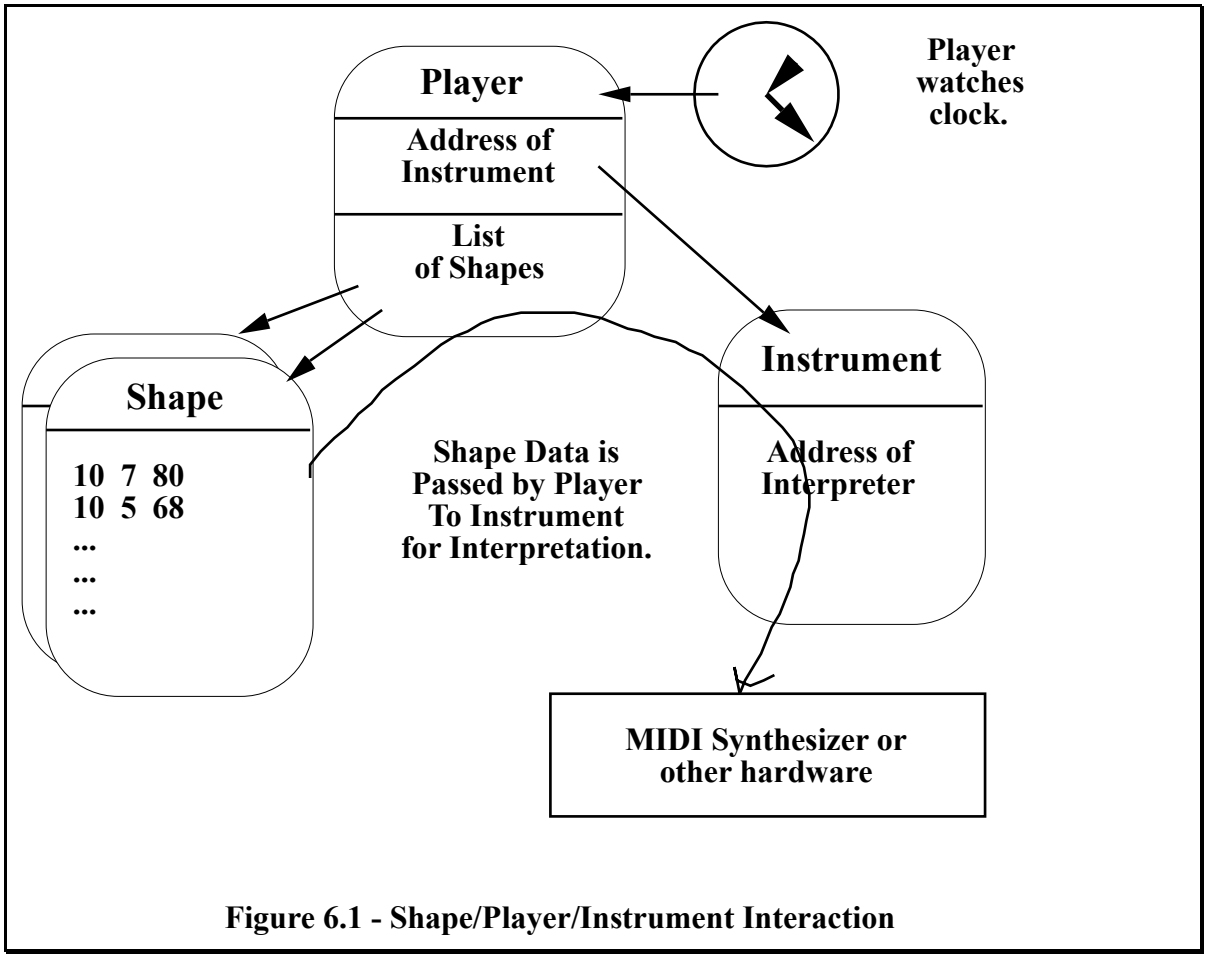


Figure 6.1 - Shape/Player/Instrument Interaction

Tutorial 1: Shapes, Players and Instruments

Shapes are played byPlayers. A player reads the timing information in a shape, typically stored in dimension 0, and sends the data to a *Virtual Instrument*, or *Instrument* for short. The instrument converts that data to something meaningful, like notes. Let's compare this to the situation where a human performer is playing. Here is a table comparing the real world objects to HMSL objects.

```
Real World      HMSL
Performer       => Player
```

```
Score          => Shape
Violin         => Instrument
```

This analogy can only be carried so far. The human performer always interprets the score perfectly (of course) and plays it on a magnificent but unchanging instrument. In HMSL, the player only interprets the timing information and leaves the interpretation of everything else to the *instrument*. The player does not even know what kind of instrument is being played. We will see later how instruments can interpret shape data as any arbitrary parameter to generate algorithmic composition, control unusual hardware, output text, or whatever the composer decides.

We supply some simple instruments with HMSL (like the MIDI Instruments, which interpret dimension 1 as pitch, and dimension 2 as velocity), but one of the most powerful features of the language is the ability for users to create their own complex and intelligent instruments to control MIDI, video, local sound, kinetic sculpture, analog hardware, or anything they want. To keep the player tutorial simple, we will use shapes that contain MIDI note information. (For examples of more exotic shape interpretation, see the chapter on Instruments and Interpreters.)

Let's instantiate a shape, a player and an instrument to experiment with. Please enter the following directly at the keyboard, not in a file. Enter:

```
OB.SHAPE  SH-1
OB.PLAYER   PL-1
OB.MIDI.INSTRUMENT  INS-1
```

We could have used predefined objects, like SHAPE-1, but it is also possible to create/instantiate as many new objects as you want. Since we studied shapes in another chapter, we won't explain how to put data in a shape. In fact let's be real lazy and let HMSL fill our shape with reasonable data. PREFAB: will fill a shape with data that follows a random walk. Enter:

```
PREFAB: SH-1
PRINT: SH-1
PRINT: PL-1
```

Notice that the player has an instrument of 0, or *no* instrument. We must tell the player to use a particular instrument. (This is like telling a performer which violin to play.) To do this enter:

```
INS-1  PUT.INSTRUMENT: PL-1
PRINT: PL-1
```

Notice that the instrument is now set but the player has no reference to SH-1. We must put the shape in the player. Enter:

```
1  NEW: PL-1   ( make room for 1 shape )
SH-1   ADD: PL-1
PRINT: PL-1
```

We could have put several shapes in PL-1. They would be played one after the other. We are now ready to play the shape. Enter:

```
PL-1    HMSL.PLAY
```

HMSL.PLAY takes a single morph on the stack, starts it and runs HMSL. You may have noticed that the shape only played once. This is because PL-1 has a repeat count of 1. Enter:

```
PRINT:   PL-1    ( notice repeat count )
10000  PUT.REPEAT:  PL-1  ( plenty long )
PRINT:  PL-1
PL-1  HMSL.PLAY
\ then quit
```

NEW: , ADD: and PUT.INSTRUMENT: are the most direct methods used in connecting shapes, players and instruments Since this is a very common activity, however, we have provided a few shortcuts. There is a quick method, called **}STUFF:** for stuffing things into objects It will automatically do a NEW: and ADD: the things you specify. As an example, enter:

```
STUFF{  SH-1  SHAPE-1  SHAPE-2  }STUFF: PL-1
PRINT: PL-1
```

This stuffed two predefined stock morphs plus our SH-1 into PL-1. You can put as many things as you want between STUFF{ and }STUFF:. If you have just one shape, and want to put it, and an instrument, in a player in one shot, you can use BUILD:. Enter:

```
SH-1  INS-1  BUILD:  PL-1
```

Build has the following stack diagram:

```
BUILD:  ( shape instrument -- , put these in player )
```

Tutorial 2: Controlling Players

This tutorial will work best if you use a sustaining MIDI preset. Reed or brass sounds will work great. Avoid percussive sounds like drums, etc. Set up your synthesizer so it responds to MIDI channel 1.

For this tutorial, let's use the shape, player, and instrument we created in the last tutorial. If you are starting fresh, and are not continuing directly from the previous tutorial, enter (not in a file):

```
OB.SHAPE  SH-1
OB.PLAYER   PL-1
OB.MIDI.INSTRUMENT  INS-1
PREFAB:  SH-1
SH-1  INS-1  BUILD:  PL-1
10000  PUT.REPEAT: PL-1
```

In this tutorial we will run HMSL in a special way. Using the word **HMSL.START**, the keyboard will remain active while HMSL is running. In an earlier chapter of this manual, we referred to this as *HMSL+Forth Mode*. This will allow us to change values in our morphs, print morphs, etc. while they are playing. You might be tempted to run like this all the time but printing and keyboard entry will be slower than normal. To start HMSL running "in the background" without the graphics opening up, enter:

```
HMSL-GRAPHICS OFF
HMSL.START
```

(This is documented more fully in the chapter "HMSL Operation"). Since the HMSL scheduler is now running, we can play a morph just by START:ing it. Enter:

```
START: PL-1
PRINT:  PL-1
```

You should hear **PL-1** start to play. When you printed **PL-1**, it probably printed more slowly. This is because HMSL gives a higher priority to the real time scheduler than to printing. When it printed, it probably listed values of 0, zero, for its *Start*, *Stop*, and *Repeat Delays*. These delays occur when a player is started, repeated, or stopped. We can set the repeat delay while PL-1 is playing. The repeat count was originally set high, 10000, so we should have plenty of repetitions to experiment with. Enter:

```
60  PUT.REPEAT.DELAY:  PL-1
```

Every time the player gets to the end of a repetition, it will delay for 60 ticks which is usually one second.

We can also affect other player parameters. As you know, a note must generally be turned OFF some time after being turned ON. Otherwise it could sound forever which is referred to as a *stuck note*. The time between a note being turned ON then OFF is called the *on time*. The time between successive note ons is called the *duration.* A player will automatically wait some fraction of the total note duration after turning the note on, then turn it off. This fraction is known as the *duty cycle*. Notice in the printout of the player that the duty cycle is 4/5. That means that the note will be on for 4/5 of its total duration. To make the melody more staccato we can change this to 1/5. Enter:

```
1   5    PUT.DUTY.CYCLE: PL-1  ( listen )
17  23   PUT.DUTY.CYCLE: PL-1  ( listen )
```

If desired, on-times can be specified explicitly per note by storing on times in a dimension of the shape. You can instruct the player to use these values by calling the PUT.ON.DIM: method. We will examine this in more detail later. (Also see the chapter on Recording and Sequencing.)

We can also affect the *duration* of the notes. Players normally read their timing information from one dimension of the shape. By default, this is dimension 0, zero, but we can use any dimension for durations. Let's use dimension 1 for durations as well as pitch. Enter:

```
1  PUT.DUR.DIM:  PL-1
```

Notice that the rhythm has become irregular and the high notes last longer than low notes. We can specify *no* duration dimension by setting this value to -1. In this case the player will use its fixed duration. Enter:

```
8  PUT.DURATION:  PL-1
-1  PUT.DUR.DIM:  PL-1   ( now it uses 8 )
16  PUT.DURATION:  PL-1
0  PUT.DUR.DIM:  PL-1  ( back to "normal")
```

There is also a third way to specify durations. We can tell the player to use a custom function to figure out the duration for an element. This function could use data from the shape, or figure out a duration based on other data. To do this we give the player a *pointer* to the function we want it to use.

We will describe *pointers* in case you are not familiar with them. Every function in Forth resides at some location in the *dictionary*. This location has an *address*. The HMSL word **'C** (pronounced "tick C") will give us the address of a function. Enter:

```
: HI CR ." Hello!" CR ;
'C HI . ( print CFA )
'C HI  30  DUMP  ( show machine code for function )
```

The number printed is the **CFA**, or function pointer, for HI. (CFA actually stands for “code field address”). The computer machine code necessary to perform the HI function is located in memory at that address. We can execute HI by name or by address. Enter:

```
HI
'C HI EXECUTE
```

The standard Forth word **EXECUTE** takes the address left by 'C and executes the function located there. When a function pointer is given to a player, it will save it and later use EXECUTE to perform that function. The functions that are called in this manner must have a standard stack diagram that suits the situation. In this case, the stack diagram for a *duration function* is:

your.dur.func (element# shape -- duration)

Note that **your.dur.func** is not an already defined HMSL routine, but refers to *any function that the user might create for this purpose*. Any function you create must have the above stack diagram. The ELEMENT# is the element about to be played in the current SHAPE. Since the shape is passed on the stack, we must use *late*

binding to access its data. Here is an example of a duration function that reads the value in dimension 0, then chooses a random value at or below that value. Enter:

```
: RAND.DUR  ( element# shape -- duration )
   0  SWAP  ( -- element# 0 shape )
   ED.AT: []  ( -- value )
   CHOOSE  1+
;
```

We can test our function by passing the same parameters the player would. Enter several times:

```
0  SH-1  RAND.DUR  .
```

Now let's tell our player to use it. Enter:

```
'C  RAND.DUR  PUT.DUR.FUNCTION:  PL-1
```

You should hear the durations change to a more random pattern. When you want to go back to using dimension 0 as durations, enter:

```
0  ( zero)  PUT.DUR.FUNCTION:  PL-1
```

When you are done listening to this player, stop it by entering:

```
STOP: PL-1
```

When we use shapes and players, they allocate memory that we must deallocate (free) when done. Enter:

```
HMSL.STOP  ( to stop the HMSL scheduler )
FREE: PL-1
FREE: SH-1
HMSL-GRAPHICS ( turn it back on for other examples )
```

Advanced Technique: Use of Local Variables in Functions

Generally, with functions like the above **RAND.DUR** it is easier to use *local variables* in the definition, rather than stack manipulation. In the case of the above function, RAND.DUR, we can rewrite it using local variables called **element#** and **shape**. Note the use of curly brackets to signify that two local variables will be created for this word, whose values will be passed in on the stack. Anything after the double hyphen is just an ordinary comment, so this word still returns a number just as before. This is a good habit to get into, since it simplifies your code tremendously, and doesn't really sacrifice much execution speed. Try the following word, and notice that it does exactly what the previous definition did, but is a lot easier to understand!

```
: RAND.DUR  { element# shape -- duration }
   element# 0
   ED.AT: shape  ( -- value )
   CHOOSE  1+
;
```

One of the nice things about local variables is that your code involves a lot less Forth-ish (and hard to read and understand) stack manipulation, and actually starts to look like your comments! Local variables are fully described in the JForth and HForth manuals, and there are lots of examples of them in the sample pieces and source code. Get in the habit of using them, they save tremendous wear and tear on your brain cells.

Tutorial 3: Algorithmic Composition

In this tutorial we will experiment with algorithmic composition. Algorithmic composition involves the use of algorithms, processes, or "recipes," for deciding what will occur musically. For this example let's use the following simple algorithm.

1) Choose a random starting note.

2) Generate a scale upwards from that note.

3) Play the scale 4 times.

4) Repeat the entire process many times.

The question is, "How do we translate this algorithm into an HMSL program?" We know how to generate a random starting note using **CHOOSE**. The scale can be generated in a **DO...LOOP** and stored in a shape. We can play the shape repeatedly using a player. In this tutorial we will learn how to embed custom functions in a player, or any morph, to perform different algorithms.

Let's enter this tutorial in a file so that we can change it later. Refer to the machine specific supplement if you need instructions on how to edit and compile files.

At the beginning of the file, we should put a comment describing the piece. We should also use **ANEW** so that we can recompile the file many times and each time *automatically forget* the code we previously defined. The beginning of the file is also a good place to instantiate the objects we will need. Enter in the file:

```
\  HMSL Player Tutorial
\  Play ascending scales
ANEW TASK-TUT3
   ( by convention, use TASK-filename )
\ Instantiate necessary objects.
OB.SHAPE  SH-1
OB.PLAYER   PL-1
OB.MIDI.INSTRUMENT  INS-1
```

Now let's write a word to setup our shape with some default values for duration and velocity. We will make room for 32 elements in case we want to edit the shape. Since we aren't using ADD: to put in the values, we must use SET.MANY: to tell the shape it has data in it. Enter in the file:

```
: INIT.SH-1  ( -- )
   32 3 NEW: SH-1
      ( allocate memory for 32 elements )
   16 SET.MANY: SH-1
      ( force element count to 16 )
   10 0 FILL.DIM: SH-1  ( set durations to 10 )
   80 2 FILL.DIM: SH-1  ( set velocities to 80 )
;
```

If you use commands like **NEW:** or **ADD:** in a file, they should be used within colon definitions. Generally, the only things which should be outside colon definitions are statements using **VARIABLE**, **CONSTANT**, **VALUE** or **CREATE**, comments or messages, or the instantiation of objects. Since Forth is very flexible, you can, of course, do whatever you want but it will be less confusing in the long run if you follow this rule.

In this tutorial, we will use a *repeat function*. Players, like *collections, actions, jobs* and *structures* have Start, Repeat, and Stop Functions. These functions are always passed the address of the morph on the stack. Thus the stack diagram for these functions MUST be:

```
your.s/r/s.function  (  morph -- )
```

Now let's write the code to generate an ascending scale. Instead of always going up by 1, we wil go up by 1, 2 or 3. Let's use *local variables* in this word. Enter in the file:

```
:  MAKE.SCALE { player | incr note -- , make a scale in SH-1)
      3 CHOOSE 1+  -> INCR
         (  choose increment = 1|2|3 )
      24 12   WCHOOSE  -> NOTE
```

```
      ( choose a random starting note )
   MANY: SH-1   0
   DO NOTE  I  1  ED.TO: SH-1
      ( set note in dimension 1 )
      NOTE INCR + -> NOTE
      ( -- increment note )
   LOOP
;
```

The above code produces a scale. We can tell the player to execute this function when it starts and every time it repeats. Thus each repetition will have a different scale.

In this piece, we want to repeat each scale 4 times. There are several ways to do this. One easy way is to put the shape in the player 4 times using the **}STUFF:** method. Now let's setup the player.

Enter in the file:

```
: INIT.PL-1  ( -- )
   STUFF{  SH-1  SH-1  SH-1  SH-1
   }STUFF:  PL-1
   INS-1  PUT.INSTRUMENT:  PL-1
   1000 PUT.REPEAT: PL-1  ( go a long time )
\ Execute MAKE.SCALE when we start and repeat.
   'C  MAKE.SCALE  PUT.START.FUNCTION:  PL-1
   'C  MAKE.SCALE  PUT.REPEAT.FUNCTION:  PL-1
;
```

We strongly recommend having a word that will initialize an entire piece, and a word that will clean everything up. This may seem like extra typing but it will really help you organize the piece. It is also very handy when testing because you can initialize the piece then examine your objects before running it.

```
: TUT3.INIT  ( -- , set everything up )
   INIT.SH-1
   INIT.PL-1
;
: TUT3.TERM  ( -- , clean up )
   CLEANUP: PL-1
;
: TUT3.PLAY  ( -- )
   TUT3.INIT
   PL-1 HMSL.PLAY
   TUT3.TERM
;
```

Compile this file according to the instructions in the Macintosh or Amiga Manual Supplement. Before trying to run the whole piece, let's test parts of it. Enter directly at the keyboard:

```
INIT.SH-1
PRINT: SH-1
```

You should see 16 elements with 10 in dimension 0 and 80 in dimension 2. Now let's try our scale generator. Enter:

```
PL-1  MAKE.SCALE
PRINT: SH-1
```

You should see increasing values in dimension 1. If this didn't work, check your file to make sure it matches the tutorial. If those worked, try the whole piece. Enter:

```
TUT3.PLAY
```

Players — subclass of OB.JOB, OB.MORPH

Players are responsible for "playing" the data of one or more shapes in time. The first dimension of a shape is generally (but not always) treated as timing information. The value in that dimension can be either the time since the shape was started, the time until the next element is to be played, or interpreted by a user function as something else (as in the previous section which showed the used of custom duration functions). When the player decides it is time for an element of a shape to be played, it passes the element number and the shape address to an *instrument*. The instrument is responsible for interpreting the shape data and converting it to musical sound, or some other form of output.

Players thus provide the "patch" in HMSL between data and what we call the*Virtual Device Interface,* or instrument definitions. The user "puts" an instrument into a player, and in that way the abstract data of shapes is "patched" to a concrete realization of some sort. Instruments, of course, are user-definable and could include things as varied as MIDI, numbers to be used for a notated score, or graphics and video. Instruments can have a great deal of intelligence as well, they can translate data from one system to another, or operate on their data in any way the user wishes.

Players are treated in the HMSL hierarchy exactly like any other morphs, except that they can *only contain shapes*. They inherit the repeat count, nodal weight, and many methods from collections. It is important to state that players are the eventual end of most hierarchies—at some point the Polymorphous Executive usually needs to *task something in time*, so there are usually players at the bottom of any tree of morphs. Players are the "leaves" of a morph tree.

Specifying Time in different ways

As mentioned above, there are two different ways of specifying the time an element of a shape is to begin playing. One form stores *ticks until the next event*. A group of 4 notes 10 ticks apart would have times 10, 10, 10, 10. This is called *relative* time because the time is relative to the next note. The other way of specifying time is as the time *since the shape started playing*, called *absolute* time. A group of 4 notes 10 ticks apart would have the following times: 0, 10, 20, 30. You can select between the two different forms of time specification using the **USE.ABSOLUTE.TIME:** and **USE.RELATIVE.TIME:** methods for players.

These two forms of time specification have the following properties:

Relative Time = 10,10,10,10:

When you remove an element, the other elements automatically "move up" to fill in the gap, which you may or may not want. Another feature is that all the durations can be easily scaled by just multiplying the time values by a fraction. The duration of the last element determines when the shape ends. Notice that there is no way to specify a gap before the first element!

Absolute Time = 0,10,20,30:

Adding or removing elements does not affect the timing of other elements. Times must be maintained in *sorted order*. This form is handy for recording and playback, and also for scheduling Cue Lists. Notice that the first element can start anytime, not just at zero. There is, however, no way to specify silence *after the last element*!

The conversion between these time forms can be done using the shape methods **DIFFERENTIATE:** and **INTEGRATE:** . This is simple except for the problem of silence at the beginning or the end of the shape. The only way to get around this is to specify START and STOP delays in the players, or to put "rests" at the beginning or end of the shape.

The default time behavior of a player can be overridden by loading it with a *custom function*. This function is called before each element is played and returns the time until the next element. See the **PUT.DUR.FUNCTION:** method.

In summary, the duration of an element in a shape is determined by the following series of rules.

1) If a *duration function CFA is non-zero*, then it is used to determine the duration. The duration function must have a stack diagram as follows:

```
your.dur.function  ( current_element# shape -- duration )
```

2) If t*he function is zero, and if the duration dimension is zero or greater* as set by **PUT.DUR.DIM:** , then the duration is taken from the duration dimension.

3) If the *CFA for the duration function is zero, **and** the duration dimension is negative*, the constant duration set by **PUT.DURATION:** will be used.

This scheme gives the user a very flexible way of controlling the time alloted to the elements of the shape being played.

Duty Cycle of Players

When elements are played, they are turned *on* and then later turned *off*. If the element is a note than this would correspond to a NoteOn and a NoteOff. Some elements are singular events, for example MIDI program change, so nothing happens when they are turned OFF. Players decide when to turn an element *off* based on several factors. A given dimension can be specified to contain *on times* for each element, as in the Type 3 shapes described in the shape chapter. When an element is turned *on*, the player will read that dimension and turn the element *off* that specified amount of time later.

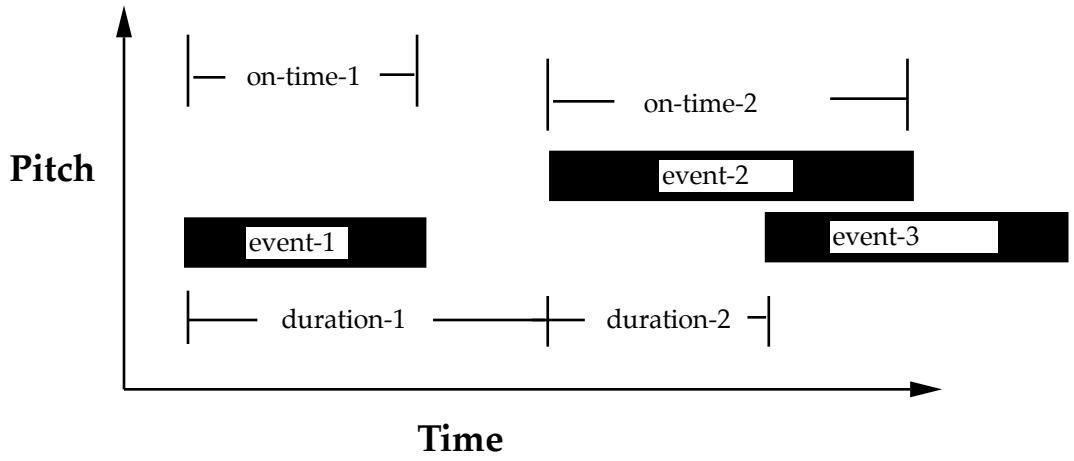
A global *duty cycle* can be specified, by setting the *on dimension* to -1. The duty cycle is the ratio between an element's *on* time and the time until the next element begins. If the duty cycle is 1/1, an element will remain on until the next element begins. This is useful for simple monophonic melodies. By modifying the duty cycle as a piece plays, *staccato or legato* playing styles can be varied.

In other words, duty cycles specify the ratio of "note-on" time to the total duration of a given element. Duty-cycles are particularly useful for MIDI information, where noteon/noteoff is often closely related to envelope. However, the duty cycle, like many of HMSL's features, is intended to be highly general and programmable so that it might be applied to a wide range of other musical ideas.

If the ratio exceeds 1:1 (i.e. the "on" time exceeds the total duration of the event) then the Polymorphous Executive will introduce a "legato" between that event and the one following. Refer to the discussion on OB.PLAYER methods below for more information on this.

Alternatively, *one of the dimensions of a shape* may be interpreted as the "on-time" for each event in the shape that is played. In this mode each event can have its own on/off ratio associated with it, which may be entered and edited as with any other dimension. The *on* dimension can be specified using **PUT.ON.DIM:** . If the dimension is set to -1 then the global duty cycle set using **PUT.DUTY.CYCLE:** will be used.

The *on* time is passed to interpreters by setting the value ON.TIME. See the Instruments and Interpreters chapter for more information on this.



"Duration" versus "On-Time"

Figure 6.2

Stock Players

PLAYER-1 thru PLAYER-4 are currently set up with INS-MIDI-1 thru INS-MIDI-4 by STOCK.INIT.

OB.PLAYER Methods

Method	Stack diagram
BUILD:	(shape instrument -- , quick startup)
DEFAULT:	(-- , set default values)
FREE:	(--)
GET.xxx:	(-- xxx , see PUT.xxx: , eg. PUT.REPEAT:)
GOTO:	(element# shape -- , jump to new position)
PLAY.ON&OFF:	(-- use both on and off interpreters)
PLAY.ONLY.ON:	(-- use just on interpreter)
PREFAB:	(-- setup "reasonable" player)
PRINT:	(--)
PUT.DUR.DIM:	(dim# -- , set which dim contains durations)
PUT.DUR.FUNCTION:	(cfa -- , set function that determines durations)
PUT.DUTY.CYCLE:	(on total --)
PUT.INSTRUMENT:	(instrument --)
PUT.ON.DIM:	(dim# | -1 -- , specifies dim. for "on" time)
PUT.REPEAT:	(repeat-count --)
PUT.WEIGHT:	(weight --)
USE.ABSOLUTE.TIME:	(-- durations absolute times of values)
USE.RELATIVE.TIME:	(-- durations relative times between values)
START:	(-- , begin execution)

```
STOP:                   ( -- , stop execution )
WHERE:                  ( -- element#  shape , what's currently playing )
```

Table 9-2. Player Methods

OB.PLAYER Methods

OB.PLAYER is a subclass of OB.COLLECTION, and OB.JOB. Please refer to the method documents for those classes.

BUILD: (shape instrument -- , quick setup)

This is a shortcut way to connect a shape, an instrument and a player. It is equivalent to:

```
1  NEW:  player
shape  ADD:  player
instrument  PUT.INSTRUMENT: .player
```

DEFAULT: (--)

Initialize player, sets duty cycle to 4:5; ON dimension is set to -1 (meaning use the duty cycle specified, not the dimensions); no instrument is specified. Dimension 0 is set for durations. Delays and Functions set to zero. Use RELATIVE time. PLAY.ONLY.ON: called.

FREE: (--)

FREE: space used by player to hold shape. Deinstantiate any objects created by PREFAB: .

GET.xxx: (-- xxx , see corresponding PUT.xxx:)

GOTO: (el# shape -- , jump to new position)

Tell the player to jump to the given element of the given shape. This can be used to skip material in the shape. See the WHERE: method. Note that GOTO: does *not necessarily* need to refer to a shape that is contained in that player, but this can be a very dangerous (if interesting) technique.

PLAY.ON&OFF: (-- , Use ON and OFF Interps.)

Make two separate calls to the instrument: once for turning the element on, then again later to turn it off. This technique is not used very often because it does not allow more than two elements to overlap in time. It also can have problems if the shape values change between the ON and OFF events.

PLAY.ONLY.ON: (-- , only use ON Interpreter)

Only call the Instrument once for each element. It will call ELEMENT.ON: which uses the ON Interpreter. This is the default.

PREFAB: (-- , setup everything needed)

This is a quick and dirty method for setting up a player. It dynamically instantiates an instrument, and a shape then calls the BUILD: method. It also PREFAB:s the shape.

PRINT: (--)

Prints player's shape names, weight, instrument, duty cycle, duration, dimension, on dimension, repeat count, scheduling type (epochal or durational), and if epochal, its toolate value. Also prints the Start/Repeat/Stop Delays and Functions.

PUT.DUR.DIM: (dimension# -- , set which dim contains durations)

Specify which dimension of all component shapes in a player contain durations or absolute time. You must ensure that the dimension# does not exceed the maximum number of dimensions in any of the shapes being played. The duration dimension is typically 0.

PUT.DUR.FUNCTION: (cfa | 0 -- , set duration function for player)

Here is an example duration function that scales the existing durations by the user data value.

```
: SCALE.DUR  ( element# shape -- duration )
   0 SWAP ED.AT: []
      ( get duration from dimension 0 )
   CURRENT.OBJECT
      ( -- dur player , get player address )
   GET.DATA: []   ( -- dur data , get user data )
   *   ( -- dur' , scale duration )
;
3 PUT.DATA: PL-DNV   ( set scale factor )
'C SCALE.DUR PUT.DUR.FUNCTION: PL-DNV
```

PUT.DUTY.CYCLE: (on total --)

The user specifies the ratio of "on time" to "total" duration for a player. For example, if a total duration is 10 for a given element of some shape, and the duty cycle is 4:5 (4 5 PUT.DUTY.CYCLE:), the "noteoff" event will be sent at 8.

PUT.INSTRUMENT: (instrument --)

Tell the player which instrument to use when playing its shapes.

PUT.ON.DIM:
(dim# | -1 -- , specifies dimension for "on" time)

Tell the player that one of the dimensions of the shape contains ON Times. ON Times are mainly used when playing notes. The ON Time is the time between Note ON and Note OFF. By storing ON Times in the shape and specifying ON Times longer then Durations, we can make notes overlap. This allows us to play chords or polyphonic sequences. A value of -1 signifies that the player's duty cycle will be used for calculating *ON* to *OFF* time.

PUT.REPEAT: (repeat-count --)

Same as in collections and other morphs. Note that if you want something to repeat many times, it's best to do it at the lowest possible level (players). This will not invoke the Polymorphous Executive more than it needs to. That is, the morphs in the tree won't have to pass a lot of *done* messages up the tree. (In other words, if you just want to hear a shape ten times, put it in a player, and give the player a repeat count of ten).

Note that like other morphs, a *zero repeat count disables* the player.

PUT.WEIGHT: (weight --)

The weight used by the *Markov Chain behavior* of Structures. It is also available to the user for other purposes, and can be a handy place for simply storing a number or some miscellaneous data. See Collections and Structures.

USE.ABSOLUTE.TIME: (-- , since beginning of shape)

The times in the shape are since the shape started playing.

USE.RELATIVE.TIME: (-- , between elements)

The times in the shape are relative to the start of the next event.

WHERE: (-- element# shape , what's currently playing)

Returns the element# and the shape that the player is currently executing. If used in conjunction with GOTO: , very interesting results can be obtained!

Advanced Topics

Player Methods That Are Internal to HMSL

v *Note:* The following methods are used by the PE (polymorphous executive) for tasking players. They are not intended for general use.

ABORT: (-- , Abort execution of player)

When HMSL.ABORT is called internally, an ABORT: message is sent to every player in the Active Object list. This results in the current element being turned off. The instrument is closed and the player sends an ABORT: message to whichever morph executed it. The ABORT: message thus propagates up the morph tree.

TASK: (-- , play pending elements)

Basic routine for tasking players, that is, deciding when to play elements from their component shapes. Turns elements off or on, and keeps track of which shape is currently playing.

?DONE: (-- done?, is the morph done)

Checks REPEAT COUNT, and when shape(s) have been completely played, sends "done" message and resets the player.