Chapter 10
Jobs & Productions

Most Important Information

Jobs provide the capability for *parallel tasks* in HMSL. They contain CFAs (addresses) of *user functions*. The time between executions is controlled using **PUT.DURATION:** . They are stopped using **SET.DONE:**. Jobs provide a mechanism for *background processing* in HMSL. They can be used to spit out notes, perform events that evolve slowly over time, etc. Jobs can have an instrument for use by the job functions if needed.

Productions are used to hold and execute user written functions. These allow you to perform custom operations at some point in the hierarchy. Productions can be put in collections. Productions contain CFAs which are obtained using **'C** followed by the name of the function.

Tutorial 1: Periodic Function

The following is an example of a simple job definition. Let's define a job that "barks" periodically. Enter the following in a file so you can experiment with it:

```
ANEW TASK-JOBTUT
\ Declare job.
OB.JOB JOB-DOG
\ Define job function.
: DO.DOG { job -- }
   ( don't need JOB address for this function, but it's stored )
   ( in the local variable anyway: good habit to get into! )
   ." Bowowowowowowow!" CR
;

: DOG.INIT   ( -- , initialize job )
   \ Put functions in job.
   STUFF{ 'C DO.DOG }STUFF: JOB-DOG
   \ Set duration.
   100 PUT.DURATION: JOB-DOG
;
```

Compile the file then enter these at the keyboard.

```
DOG.INIT   ( perform initialization )
PRINT: JOB-DOG
JOB-DOG HMSL.PLAY  ( now watch text appear in Forth window )
```

Notice that you can use the Shape Editor while the job is running. The whole point of using Jobs is that they run in parallel with other parts of HMSL. Quit HMSL and enter:

```
FREE: JOB-DOG
```

Tutorial 2: Notes from Jobs

To demonstrate that jobs can also be used for musical purposes let's define a job that plays notes slower and slower. Add this to the previous file:

```
\ Play random notes slower and slower.
OB.JOB MY-JOB

: MYJOB.FUNCTION  { job -- , play random note }
   MIDI.LASTOFF
   JOB GET.DURATION: [] ( --  duration )
   1+  DUP ( -- duration+1 duration+1 )
   JOB PUT.DURATION: []   ( -- duration+1, make duration bigger )
   50 > ( -- )
   IF
      JOB SET.DONE: [] ( terminate job )
         ( reset duration for next time thru  )
      0 JOB PUT.DURATION: []
   ELSE
      90 36 WCHOOSE 64 MIDI.NOTEON
   THEN
;
\ Separate INIT and TERM for easier testing.
: MYJOB.INIT  ( -- )
   STUFF{ 'C MYJOB.FUNCTION }STUFF: MY-JOB ( put one in )
   0 PUT.DURATION: MY-JOB ( initialize duration )
;
: MYJOB.TERM ( -- , clean up )
   FREE: my-job
;
: MYJOB.PLAY  ( -- )
   MY-JOB HMSL.PLAY
;
: SLOW.JOB  ( -- )
   MYJOB.INIT
   MYJOB.PLAY
   MYJOB.TERM
;
```

Compile the file, then enter directly:

```
SLOW.JOB
```

which should take about a minute before it stops. You can change the repeat-count of this job, which will determine how many times the entire process executes (not just one iteration of the function). That is, if the repeat-count is 2, then the job will finish once, and be "reincarnated" once. Note that though **SET.DONE:** is called from inside the function, the job will finish its current execution before telling its parent that it is through.

Now let's combine the two previous examples to show how jobs run in parallel. Add to the end of the above file:

```
   \ Combine two Jobs
OB.COLLECTION  COLL-BOTH
: BOTH.INIT  ( -- , combine jobs )
   DOG.INIT
   MYJOB.INIT
   STUFF{  JOB-DOG  MY-JOB  }STUFF: COLL-BOTH
   ACT.PARALLEL:  COLL-BOTH
```

```
;
: BOTH.TERM  ( -- )
   FREE: JOB-DOG
   MYJOB.TERM
   FREE: COLL-BOTH
;
: RUN.BOTH  ( -- )
   BOTH.INIT
   PRINT.HIERARCHY: COLL-BOTH
   COLL-BOTH  HMSL.PLAY
   BOTH.TERM
;
```

Compile the file and enter:

```
RUN.BOTH
```

Tutorial 3: MIDI Modulation Changer

Here is an example of a job that randomly changes the MIDI modulation depth every 5 seconds. Enter this in a file:

```
ANEW TASK-JOBTUT3
OB.JOB PR-CHANGER

: CHANGE.DEPTH  { job -- , change modulation depth }
\ This function will be called by the JOB
\ Controller 1 is modulation depth.
    1  128 CHOOSE MIDI.CONTROL
;

: SETUP.JOB  ( -- , setup job for example)
    STUFF{ 'C CHANGE.DEPTH }STUFF: PR-CHANGER
    300 PUT.DURATION: PR-CHANGER  ( 5 seconds )
;

: DOIT  ( -- , do whole thing )
    SETUP.JOB
    PR-CHANGER  HMSL.PLAY  ( use it )
    FREE: PR-CHANGER
;
```

Now compile the file and enter:

```
DOIT
```

In this example, the job could be used to affect a synthesizer while it is being played. One could also place this job in a collection in a hierarchy so that one section of the piece would be affected whenever it ran.

Jobs

Jobs are morphs that can contain a *list of functions* to be executed at *specific time intervals*. When a job (or player) is executed by its parent, it posts itself to the multitasker. The multitasker then sends **TASK:** messages to the job as often as possible. The job then decides whether to execute its list of functions depending on whether enough time has elapsed since the previous time.

This differs from *productions* which execute their list of functions *immediately when executed* by their parent. Productions do not post themselves to the multitasker. Thus, jobs are useful for activities which are prolonged in time, while productions are useful for once only activities.

The class OB.JOB is a subclass of OB.PRODUCTION (and thus OB.COLL.SEQ). By using the NEW: and ADD: methods, you can put as many functions as you want into a job.

The frequency of execution of the job can be controlled by setting a *duration* specific to each job. Jobs are the most convenient place to put a function that you want executed at specific time intervals.

Players are a subclass of jobs, and as such, all of the methods that are defined for jobs are used in players as well. In fact, many of these methods are more often used in players (like those that apply to instruments).

Jobs may be put in collections, or structures. However, note that if a job is put in a sequential collection or structure, some way for the job to terminate must be programmed into the system. Usually this is done using the job's own SET.DONE: method (see below).

The JOB Function

User-written functions are placed in a job in the same way that they are put in productions (see the chapter on productions). The stack diagram for these functions is simple:

```
( job_address -- )
```

You may use any function(s) as long as they conform to this stack diagram.

The argument is the job address itself. This is analogous to the way instrument interpreters are passed the address of the instrument, and other such protocols in HMSL. By using this address, the job function may make use of the methods and instance variables of the job in which it "resides." It is highly reccommended that you get into the habit of passing this address immediately into a local variable, probably called JOB.

To change its own duration, or to terminate itself, the job function should use this address to execute one of its appropriate methods (GET and PUT.DURATION: , SET.DONE:). If the job does not need to stop (for example, if it is a background process for a whole piece), then the function does not need to have a way to stop it.

Instance variables for the class include ones for duration, instrument, repeat-count, weight, and to decide whether or not epochal or durational scheduling is used (see players).

Job Methods

Method	Stack diagram
ABORT:	(--)
DEFAULT:	(-- , set default values)
GET.DURATION:	(-- duration)
GET.INSTRUMENT:	(-- instrument)
GET.REPEAT:	(-- repeat-count)
GET.TOO.LATE:	(-- #ticks, returns maximum lateness allowed)
PLAY:	(--)
PRINT:	(--)
PUT.DURATION:	(duration --)
PUT.INSTRUMENT:	(instrument --)
PUT.REPEAT:	(repeat-count --)
PUT.TOO.LATE:	(#ticks -- , set maximum lateness allowed)
SET.DONE:	(--)
STOP:	(-- , stop tasking)
USE.DURATIONAL:	(--)
USE.EPOCHAL:	(--)

Table 9-1. Job Methods

ABORT: (--)

STOP:s the job, and "aborts" up the tree. (See chapter on morphs.)

DEFAULT: (--)

Default settings for a job are epochal scheduling (about five minutes for the "too late window"), no instrument in use, and a duration of zero. A duration of zero means that the job will happen "as often as possible".

GET.DURATION: (-- duration)

Returns the duration of a job. See PUT.DURATION: .

GET.INSTRUMENT: (-- instrument)

Returns instrument address.

GET.REPEAT: (-- repeat-count)

Returns repeat-count of job.

GET.TOO.LATE: (-- #ticks, returns maximum lateness allowed)

PLAY: (--)

Used for testing, executes the job once.

PRINT: (--)

Prints duration, scheduling type, instrument, and names of functions in job.

PUT.DURATION: (duration --)

Specify the duration of a job, or the time interval of execution. Can be used while a job is executing.

PUT.INSTRUMENT: (instrument --)

Assign an instrument to a job. This instrument can be used by a job function but otherwise is not necessary. The job will only open and close it. It is not automatically used in a job like it is in a player. (For more details, see the chapter on instruments and the discussion on players.)

PUT.REPEAT: (repeat-count --)

Function(s) of job will be executed repeat-count many times each time the job is TASKed.

PUT.TOO.LATE: (#ticks -- , set maximum lateness allowed)

Specify "too late window" for epochal scheduling. Same as for players.

SET.DONE: (--)

When the job is finished with the current function, it will tell its parent that it is done. This method is essential, as it is the only way to terminate a job, in, for example, a sequential collection, and to move on to the next morph! If the job does not need to terminate, the job's function does not need to call SET.DONE: .

START: (-- , start a job)

This must be called when the HMSL Scheduler is running. This is equivalent to:

```
TIME@  0  EXECUTE:
```

STOP: (-- , stop tasking)

STOP:s execution of the job.

USE.DURATIONAL: (--)

Use durational scheduling in the job (same as for players).

USE.EPOCHAL: (--)

Use epochal scheduling in the job (same as for players).

Productions

Productions are the superclass for OB.JOB and OB.PLAYER.

Productions provide a way for you to embed custom functions inside a hierarchy. Productions are a subclass of collections whose members do not contain executable morphs. Instead, they contain a list of executable Forth routines as their children. Productions can be contained in other morphs (collections, structures, etc.), and can be executed like a collection or structure.

Most of the things you can do with a production can also be done using START, STOP and REPEAT functions in other morphs.

Productions must be NEW:ed like collections, for example:

```
OB.PRODUCTION MY-PRODUCTION
2 NEW: MY-PRODUCTION
```

Productions contain no new methods. Functions are added to a production by the following syntax:

```
'C FOO ADD: MY-PRODUCTION
```

Functions that are placed in productions must have a stack diagram as follows:

```
FOO  ( -- , nothing taken or left behind )
```

PUT: , GET: , and other methods associated with repeat count and weight are inherited from collections.

Productions, when executed, execute their component list of CFAs. They are not "tasked," so they will seize control of the system for the time it takes them to execute their component routines. Care must be taken that these routines are not too consumptive of the CPU, or time deformations will occur.

Productions are extremely simple and extremely powerful. They can be used to transform shapes in real time, generate data in any form, or alter any part of the system.

When a production is executing its functions, the variable CURRENT-PRODUCTION contains the address of the production.