Chapter 7
Instruments and Interpreters

Most Important Information

The most important methods for controlling instruments are **PUT.OFFSET:** , **PUT.GAMUT:** , **PUT.CHANNEL:** and **PUT.CHANNEL.RANGE:** . Interpreters are important in customizing the way that an instrument interprets raw shape data. The default interpreter treats dimensions 1 and 2 as note and velocity for melodies. An important technique to learn in this chapter is how to write a custom interpreter that *conforms to the required stack diagram* and how to put interpreters in instruments using **PUT.ON.FUNCTION:** and **PUT.OFF.FUNCTION:** .

Introduction

Instruments provide an interface between HMSL and hardware, music or otherwise. They can be thought of as intelligent "device drivers." HMSL provides *3 predefined Instrument classes*. The first is a *generic instrument* which is mainly used to build other specialized Instrument classes. The **OB.MIDI.INSTRUMENT** is designed to support MIDI output. The **OB.AMIGA.INSTRUMENT** drives the Amiga internal 4 voice, 8-bit sample playback system.

One can define a custom instrument class for a specific hardware device. An interesting example is the OB.BELL.INSTRUMENT that Phil Burk wrote to interface HMSL to composer David Mahler's *Washington State Bell Garden*. This instrument controlled bells through solenoids connected to the Amiga's parallel port. By substituting Bell Instruments in place of the MIDI Instruments, Mahler and Burk were able to run existing HMSL pieces on the Bells.

Players and jobs can have their own instruments which they open when executed and close when done. Jobs don't use the instrument themselves but do make it available for the *job function* in case it needs an instrument. Players, on the other hand, use their instruments to play their shapes. The player decides when to play the elements of its shapes based on its particular scheduling algorithms. When it is time to play a shape element, it is passed to the instrument. The instrument then interprets this raw shape data and converts it into some meaningful form of output. This interpretation is carried out by *interpreters* which are customizable Forth routines that are used *inside instruments*.

One can *write one's own interpreters* for instruments or use the existing ones. The default interpreter treats dimension #1 of a shape as a note index and dimension #2 as velocity. When this note is played, it is translated to a MIDI note using an optional *translator* object and an offset. The translator contains a *gamut* of notes to be used. This allows the composer to restrict the notes, for example, to a "major key," whole tone scale, or anything else. The gamuts are customizable and can even be represented by a function for "algorithmically evolving keys."

We could write a custom interpreter that treated an added shape dimension as MIDI preset or controller values (one of the demos on the disk does this). A more abstract example would be to interpret the shape data as parameters for an algorithm that is generating notes based on a mathematical formula.

A predefined HMSL interpreter (**INTERP.EXECUTE**) is available that considers one dimension of the shape to be the *address of a function*, and two of its dimensions as parameters. This can be used to schedule the execution of various user functions at specific times. Another predefined interpreter (**INTERP.PLAY.MORPH**) treats one dimension as the address of morphs which can be started or stopped at various times. Some very complex scheduling can be achieved using this system.

Interpreters can also been written that treat shape data as graphical information. This allows animation effects to be scheduled along with the music. On the Amiga, the animation toolbox of JForth can be called from HMSL. (See **HP:PHASEPIX** on the Amiga) The output of the Amiga can be recorded directly onto video tape, making it possible to produce interesting musical animation pieces.

Another function of instruments is to keep track of which notes are currently sounding. They can then turn them off when needed. If a player is stopped, for example, it can send an ALL.OFF: command to its instrument which then turns off all its notes. This is to avoid having hung notes.

Instruments can also dynamically reserve a MIDI or Amiga channel for their use. Instruments can be forced to use a specific channel but sometimes this is undesirable. Imagine an algorithmic piece where there are 30 different players of which up to 8 might be playing at any given time. It may be impossible to predict which players will be playing together at any given time. It is, therefore, impossible to assign channels arbitrarily to these players because they are likely to conflict. With a multitimbral device, like an 8-channel Yamaha FB-01, one can tell the instruments to allocate a channel from a given range when they are used. They can then select the preset they need. The channels on most multitimbral synths are equivalent, so it usually doesn't matter which one is used as long as there are no conflicts. HMSL has a central *MIDI channel resource allocator* that keeps track of which channels are in use throughout the system. This makes the sharing of limited channel resources between independant "processes" quite transparent.

Tutorial 1 - Using Instruments Directly

Required: Polyphonic MIDI Instrument assigned to MIDI channel 1. Use a sustaining voice like a brass or reed sound instead of a percussive sound.

Instruments are normally used indirectly by putting them in players and letting the players deal with them. There are times, however, when it is important to know how to use instruments directly. To learn how instruments work, let's instantiate a MIDI instrument. (Enter this tutorial directly at the keyboard, not in a file.) Enter:

```
OB.MIDI.INSTRUMENT  INS-1
PRINT: INS-1
```

Notice that this instrument has a *channel range*. One of the functions of an instrument is to *allocate a channel* to play on. You can specify a range of channels using **PUT.CHANNEL.RANGE:** if you want it to allocate between specific channels. The default for a MIDI instrument is 1 to 16. Notice that the current channel is (-1). This is because no channel has been allocated yet. An instrument *allocates its channel when it is opened*. Let's open the instrument and see the result. Enter:

```
OPEN:  INS-1
PRINT:  INS-1
```

Notice that the channel is now set to 1 (assuming you have not allocated anything previously). If you immediately opened another instrument, it would allocate channel 2. The channels are allocated from a list called the **MIDI-ALLOCATOR**. Enter:

```
PRINT:  MIDI-ALLOCATOR
```

Note that channel 1 has been marked as "In Use" by the being set to 1. This allocation scheme is useful when you have lots of instruments opening and closing in a complex way. It would be very hard to predict what channels would be available at a given time. This way, as long as a channel is available, it will get allocated. If a channel is not available, it will just pick one and use it anyway. You can *force* an instrument to use a specific channel using the **PUT.CHANNEL:** method.

Now that the instrument is open, we can use it. One of the primary methods of an instrument is to *turn notes on and off*, and to keep track of which ones are on. The instrument will play on whatever channel it is set to. Enter:

```
12  80  NOTE.ON:  INS-1
PRINT: INS-1
```

You should have heard a low C begin playing. Notice at the top of the instrument that the number 48 is displayed. This means that this instrument is currently playing MIDI note 48. If we entered 12, why did it play 48?

Notice that the instrument has an *offset of 36*. This offset is *added* to each note index before being played. This allows simple transposition and also hides some of the peculiarities of MIDI from the user. Most MIDI synthesizers have note 36 as their lowest note, a very low C note. By adding an offset in the instrument, our programs can use numbers that start at 0 or 1 which feels more normal. If our piece used numbers above 36, it might not work on a synthesizer that was not MIDI based. David Mahler's *Bells*, for example, were numbered from 0 to 28 so the *Bell Instrument* had a zero offset. This offset allows us to program in a more abstract way.

You are probably tired of hearing that note so let's turn it off. Enter:

```
12  0  NOTE.OFF:  INS-1
```

We can change the offset any time we want. Enter:

```
50  PUT.OFFSET:  INS-1
12 80 NOTE.ON:  INS-1   ( much higher this time )
12 0 NOTE.OFF: INS-1
```

There are several different ways to turn off the notes. Since the instrument keeps track of which notes are playing, we can turn them all off at once. Enter:

```
5  70  NOTE.ON:  INS-1
9  75  NOTE.ON:  INS-1  ( hear two notes together )
ALL.OFF:  INS-1  (  silence )
```

If you did not hear two notes playing together, check to make sure your synthesizer is in a polyphonic mode. You can also turn off notes based on the order they were played. (When entering this example, don't forget about the command line history feature using the cursor arrows.) Enter:

```
7   70  NOTE.ON:  INS-1
11   70  NOTE.ON:  INS-1
15   70  NOTE.ON:  INS-1  ( 3 at once )
PRINT: INS-1
FIRST.NOTE.OFF:  INS-1
PRINT: INS-1
LAST.NOTE.OFF:  INS-1
PRINT:  INS-1
LAST.NOTE.OFF:  INS-1
```

You can also have notes turn themselves off automatically after a certain time. This technique takes advantage of the *Event Buffering System* described in the chapter on *Time and Scheduling*. To make the system more responsive to our input, enter:

```
10  TIME-ADVANCE  !
```

Now the *main HMSL clock* will run 10 ticks ahead of the *Real Time Clock*. Enter on one line:

```
TIME@  VTIME!   4  80  30  NOTE.ON.FOR:  INS-1
```

The first two commands set the *Virtual time to now*. Then we turned on note 4 with velocity 80 for 30 ticks. You should have heard a note sound for about half a second. When an instrument is being played by a player, the player sets the *virtual time to the time the note should be heard*. Here is a simple word that plays two notes together then one note later using NOTE.ON.FOR:. Enter:

```
:  PLAY3  (  time  -- )
   VTIME!  ( set starting time )
   5  80  30  NOTE.ON.FOR:  INS-1  ( play two together )
   12  90  40  NOTE.ON.FOR:  INS-1
   60  VTIME+!  ( advance virtual time by 60 )
   15  80  60  NOTE.ON.FOR:  INS-1
;
TIME@  PLAY3
TIME@  200  +  PLAY3  ( play them later )
```

When we are done using the Instrument we must close it. Enter:

```
CLOSE: INS-1
PRINT: INS-1  ( note channel = -1 )
PRINT:  MIDI-ALLOCATOR
FORGET  INS-1
```

Note: If your program crashes, it may not deallocate the MIDI channels. This may make it appear that there are no more MIDI channels available. If this occurs you can enter:

```
CLEAR:  MIDI-ALLOCATOR  ( to make all channels available )
```

Tutorial 2 — Using Instruments with Players

Required: Same as tutorial 1. Before doing this tutorial, make sure you understand the first tutorial on players.

In this tutorial we will see how instruments are used by players to play shapes. We will need a shape, a player, and a MIDI instrument. You may have some already defined but since they may be in an unknown state, let's make some fresh ones. Enter (directly):

```
OB.SHAPE  SH-1
OB.PLAYER  PL-1
OB.MIDI.INSTRUMENT  INS-1
PREFAB:  SH-1  ( some fake notes )
SH-1  INS-1  BUILD:  PL-1  ( connect them together )
10000 PUT.REPEAT:  PL-1
```

Now let's start HMSL and the player. When a player starts, it automatically opens the instrument. Enter:

```
HMSL.START
START:  PL-1
PRINT:  INS-1
```

Note that the instrument is open and has an assigned channel. Let's experiment with changing the offset and hear the notes transpose. Enter:

```
48  PUT.OFFSET:  INS-1  ( transpose up an octave )
```

This instrument allows any notes in the 12 tone scale to be played. For some pieces, this may not be appropriate. If we want to restrict the notes that can be played, we can specify a **gamut**. A gamut is the set of

allowable notes. If we want a melody to be in D-Major, we can put the notes of a D-Major scale in a gamut and tell the instrument to play only those notes.

The instrument will use the number passed using NOTE.ON as an index into the gamut. Thus you could pass the notes 0,1,2,3,4,5,6,7 to the instrument and it would play the proper notes in the scale. You can use any object that supports the **TRANSLATE:** method as a gamut. The TRANSLATE: method takes an input number and translates it to another number. There is a predefined class called an OB.TRANSLATOR that works well for this purpose. Enter:

```
OB.TRANSLATOR  GAMUT-1
STUFF{  0  2  4  5  7  9  11  }STUFF: GAMUT-1
PRINT: GAMUT-1
```

This gamut now has the note relations that make up the major scale. We can transpose the gamut from a C-major to a D-major by putting an offset in it. Enter:

```
2  PUT.OFFSET:  GAMUT-1  ( was C, now D major )
0 TRANSLATE:  GAMUT-1 .  (
1 TRANSLATE:  GAMUT-1 .
2 TRANSLATE:  GAMUT-1 .
7 TRANSLATE:  GAMUT-1 .  (
```

When we translated 0,1,2 we should have gotten 2,4,6 — the first 3 notes of D-major. To do this in our instrument, enter:

```
GAMUT-1  PUT.GAMUT:  INS-1
```

The pitch went way up, because we now go up an octave for every 7 notes instead of 12. The instrument offset of 36 means we are 5 octaves above 0 instead of 3 octaves. To have the instrument start at 3 octaves above 0 we must use an offset of 3*7=21. Enter:

```
21  PUT.OFFSET:  INS-1
```

This should now be back in the same octave as originally but playing in D-major instead of a 12 tone scale. Changing offsets in the gamut and instrument do different things:

Gamut Offset determines **which key**, eg. C- versus D-major

Instrument Offset determines **transposition within key**.

To transpose the melody without changing keys, enter:

```
24  PUT.OFFSET:  INS-1
```

To change keys from D- to F-major, enter:

```
5  PUT.OFFSET:  GAMUT-1
```

If you want to find out what the actual MIDI note value will be after adding the offset and using the GAMUT, there is a TRANSLATE: method for instruments. Enter:

```
11  TRANSLATE:  INS-1  .
12  TRANSLATE:  INS-1  .
```

With the offsets we used, we got 65 and 67. You can also go the reverse direction. That is, you can *detranslate* to find out the input value for a specific MIDI note. Enter:

```
65  DETRANSLATE: INS-1  . .  ( TRUE and 11 )
```

```
66  DETRANSLATE: INS-1  .  ( FALSE , not "in key" )
```

If the MIDI note you specify is not available, DETRANSLATE: returns a FALSE. If it is available, it will return the necessary input and a TRUE.

Translators have a *modulus* which allows them to repeat the same pattern for every octave. They can also use a *function* to translate data instead of a table. Translators can be a very powerful tool in HMSL. For more information on how they work, see the chapter on translators.

Let's explore some other instrument methods. An instrument can be temporarily muted. This can be handy if you want to silence an instrument without stopping and restarting it. Enter:

```
TRUE  PUT.MUTE:  INS-1  ( silence it )
FALSE  PUT.MUTE:  INS-1  ( turn off mute )
```

You can specify a MIDI Preset (Program) using PUT.PRESET: . The preset number will be retained by the instrument and sent whenever the instrument is opened. Enter:

```
9  PUT.PRESET:  INS-1
```

To temporarily change the preset for an instrument without making it permanent, we can use PRESET:

```
14  PRESET:  INS-1
```

If we CLOSE: and the OPEN: the instrument it will go back to using preset 9.

Now let's stop and cleanup from this tutorial. Enter:

```
STOP:  PL-1
CLEANUP:  PL-1  ( also cleans up shape and instrument )
HMSL.STOP
FREE:  GAMUT-1
FORGET  SH-1
```

Tutorial 3 — Experimenting with Interpreters

When a player plays a shape, it passes the *index of the current element* and the *shape address* to the instrument. The instrument then uses a special function called an interpreter that reads the data from the shape and does something with it. The default interpreter, used in the previous examples, interprets the shape data as notes and plays them on the instrument using the **NOTE.ON.FOR:** method.

We can write special interpreters to use shape data any way we want. This is one of the most powerful features of HMSL. To better understand interpreters, let's do some experiments. Please enter this tutorial in a file. We will be changing the definition of one of the words repeatedly so it will be more convenient to have it in a file. Feel free to use lower case in the files. We use upper case in this manual to make the Forth examples stand out from the English text.

Please enter in a file:

```
\ Test Interpreters
ANEW TASK-TEST_INTERPRETERS

\ Declare Objects
OB.SHAPE  TI-SHAPE
OB.PLAYER  TI-PLAYER
OB.MIDI.INSTRUMENT  TI-INSTR

: TI.INIT  ( -- , set everything up )
   PREFAB:  TI-SHAPE  ( some fake notes )
```

```
      TI-SHAPE  TI-INSTR  BUILD:  TI-PLAYER
      TI-SHAPE  ADD: SHAPE-HOLDER
      10000 PUT.REPEAT:  TI-PLAYER
   ;
   : TI.TERM  ( -- , cleanup )
      CLEANUP:  TI-PLAYER
      TI-SHAPE  DELETE:  SHAPE-HOLDER
   ;
   IF.FORGOTTEN TI.TERM

   : TI.PLAY  ( -- , play player )
      TI.INIT
      TI-PLAYER  HMSL.PLAY
      TI.TERM
   ;
```

Save this file on disk, then compile it into HMSL. (See the Macintosh or Amiga Manual supplements if you have trouble here.) To test it, enter directly into HMSL:

```
TI.INIT  ( do set up )
PRINT:  TI-PLAYER  ( should have TI-SHAPE and TI-INSTR )
PRINT:  TI-SHAPE  ( shold have data for several notes )
TI-PLAYER  HMSL.EXEC  ( hear notes )
TI.TERM
TI.PLAY  ( hear it again )
```

HMSL.EXEC is a simple word that plays a morph until it stops, without putting up the graphics window. It is handy for simple tests. Let's write a very simple interpreter that prints out its parameters. What are the parameters for an interpreter? Good question. This is *very important*. ALL Interpreters MUST have the following stack diagram:

`any.interpreter  ( element#  shape  instrument -- )`

The ELEMENT# is the element the player has decided to play. The SHAPE parameter is the shape currently playing. The INSTRUMENT parameter is the instrument using the interpreter. Let's write a simple interpreter that just prints these out. Go back to the text editor and enter in the file (just before TI.INIT):

```
: CUSTOM.INTERP  ( element# shape instr -- )
   NAME: [] SPACE  ( instrument )
   NAME: [] SPACE  ( shape )
   .  CR  ( element# )
;
```

Notice that we had to use late binding '[ ]' in the interpreter because the objects we wanted to use were passed on the stack.

To tell an Instrument to use this special function, we must give it the address of that function (CFA). **'C** will give us the address. The method **PUT.ON.FUNCTION:** will tell the instrument to use the function whose CFA is on the stack as its *on interpreter*. The instrument will later use **EXECUTE** to execute our function. Change TI.INIT by adding this line right before the '**;**'.

```
'C CUSTOM.INTERP  PUT.ON.FUNCTION:  TI-INSTR
```

Save and compile the file. We can test this interpreter by entering:

```
2  TI-SHAPE  TI-INSTR  CUSTOM.INTERP
```

Use this technique whenever you need to test an interpreter. You should see the input parameters echoed. Now let's test it in the instrument. Enter:

```
TI.PLAY
```

You will see a line of text for each element. Note that the *index increases* as it plays each element, and that there was no sound. This is because we have replaced the default interpreter for PL-1, which plays notes, with our custom one.

Now let's write an interpreter that actually uses the shape data, but let's just print it this time. Go back to the file and change the definition to match the following:

```
: CUSTOM.INTERP  ( element# shape instr -- )
    DROP  ( don't need instrument )
    GET: []  ( -- dur note vel ,   get shape data )
    ROT  .    SWAP  .  .  CR
;
```

Recompile the file and enter:

```
TI.PLAY
```

You should see the shape data displayed. Now let's write an interpreter that uses this shape data to play notes. (We will use *local variables* in this example so be sure to use curly braces on the stack diagram! Note that you don't have to use [] with a local variable. You can bind directly to it. This will actually compile to a late binding.) Change the definition in the file to match the following:

```
: CUSTOM.INTERP  { element# shape instr -- }
   ELEMENT#  1 ED.AT: SHAPE ( NOTE )
   ELEMENT#  2 ED.AT: SHAPE ( VEL )
   ON.TIME   ( from player )
   NOTE.ON.FOR: INSTR
;
```

Recompile the file and enter:

```
TI.PLAY
```

You should hear the notes being played like you are used to hearing. This interpreter is essentially like the one we use by default. The word **ON.TIME** returns the *ontime* for the current note. It is set by the player as it analyses the timing information in the shape. The player then calls the instrument which calls the interpreter so ON.TIME always refers to the current element that the interpreter is reading. The **NOTE.ON.FOR:** method plays the note for that length of time. Here are the stack diagrams for the words we just used.

```
ED.AT:  ( element# dimension  -- value , shape method )

ON.TIME  ( -- ontime  , set by player )

NOTE.ON.FOR: ( note velocity ontime -- , instrument method )
```

Here are some more interpreters that you could use. Try substituting these definitions for the one in the file. For these you may want to use the shape editor to edit the melody while it is playing to make the effect more apparent. Select the shape by clicking on the "up arrow" in the Shape Selector then click on the name TI-SHAPE. Also turn on the tracking by clicking on the "Track" button. (The lines that are unique to each interpreter are in bold face.)

This first one plays a *random note* up to the value specified in the shape.

```
: CUSTOM.INTERP  { element# shape instr -- }
   ELEMENT#  1 SHAPE ED.AT: []  ( NOTE )
   CHOOSE  ( pick random # )
```

```
   ELEMENT#  2 SHAPE ED.AT: []  ( VEL )
   ON.TIME   ( from player )
   INSTR  NOTE.ON.FOR: []
;
```

CHOOSE takes the note value and returns a random number between ZERO and NOTE-1.

This next one plays the notes *inverted around the first note*.

```
: CUSTOM.INTERP  { element# shape instr -- }
   0  1 SHAPE ED.AT: []  ( 1st Note )
   ELEMENT#  1 SHAPE ED.AT: []  ( note )
   -  ( subtract from 1st to invert )
   ELEMENT#  2 SHAPE ED.AT: []  ( vel )
   ON.TIME   ( from player )
   INSTR  NOTE.ON.FOR: []
;
```

This next one uses dimension 2 as a MIDI *preset* value. The velocity is set to 64 which is the default for MIDI. You may have to lower the values in dimension 2 using the Shape Editor if your synthesizer does not have high numbered presets.

```
: CUSTOM.INTERP  { element# shape instr -- }
   ELEMENT# 2 SHAPE ED.AT: []  ( preset )
   INSTR  PRESET: []
   ELEMENT#  1 SHAPE ED.AT: []  ( note )
   64 ( use default velocity )
   ELEMENT#  2 SHAPE ED.AT: []  ( vel )
   ON.TIME   ( from player )
   INSTR  NOTE.ON.FOR: []
;
```

Here are some suggestions for Interpreters that you may enjoy writing:

1) Send shape data to a synthesizer using MIDI System Exclusive commands.

2) Call RTC.RATE! with shape data to change the master tempo.

3) Use a dimension for PITCH.BEND information to play a monophonic microtonal melody on a synthesizer that does not directly support alternative tunings.

We recommend looking at some of the examples pieces for ways of using interpreters. Here is a list of the relevant ones.

HP:DEMO_PRESET - uses dim 3 as MIDI preset

HP:BOOKS - interprets shape as abstract parameters

HP:DEMO_INTERPRETER - one shape transposes another

HP:DEMO_MANY - play several notes at once

HP:SCHEDULE_MORPHS - uses INTERP.PLAY.MORPH

HP:SPLORP - records and plays back data for algorithm

HP:DEMO_CHORDS - shape has root, chord type, etc.

The following use Amiga local sound:

HP:SQUISH - shape contains tuning information

HP:SWIRL - play Amiga periods and sample index

HP:PHASEPIX - play notes with animation

HP:DEMO_WAVE - play with random offset

OB.INSTRUMENT subclass of OB.LIST

This is a generic instrument class that is normally used to build more useful instruments like OB.MIDI.INSTRUMENT and OB.AMIGA.INSTRUMENT. The other instruments inherit these methods so you should be familiar with them.

Instruments inherit the methods of OB.LIST. This allows them to keep track of all the notes that have been turned on. This is useful because these notes must eventually be turned off or they will sound forever. When you PRINT: an OB.INSTRUMENT you will see all the notes that have been turned on. Note that they have already been translated so that values may not look familiar to you. This list is used by the ALL.OFF: , LAST.NOTE.OFF: and FIRST.NOTE.OFF: methods.

Method	Stack diagram
CLOSE:	(--)
DEFAULT:	(-- , set default values)
ELEMENT.OFF:	(elmnt# shape --)
ELEMENT.ON:	(elmnt# shape --)
GET.CHANNEL:	(-- channel | -1)
GET.CHANNEL.RANGE:	(-- lo hi)
GET.GAMUT:	(-- gamut)
GET.OFFSET:	(-- offset)
GET.TUNING:	(-- tuning)
NOTE.OFF:	(note_index velocity --)
NOTE.ON:	(note_index velocity --)
OPEN:	(-- , open for use)
PRINT:	(--)
PUT.#VOICES:	(#voices -- , set maximum #voices for polyphony)
PUT.CHANNEL:	(channel | -1--)
PUT.CHANNEL.RANGE:	(lo hi -- , set allowable range)
PUT.CLOSE.FUNCTION:	(cfa --)
PUT.GAMUT:	(gamut --)
PUT.OFF.FUNCTION:	(cfa --)
PUT.OFFSET:	(offset --)
PUT.ON.FUNCTION:	(cfa --)
PUT.OPEN.FUNCTION:	(cfa --)
PUT.TUNING:	(tuning --)
TRANSLATE:	(note_index -- actual_note)

Table 12-1. Instrument Methods

ALL.OFF: (-- , turns all notes off)

This method turns off all the notes turned on using NOTE.ON: . This is handy if you have stuck notes or just want a simple way to shut everything off.

CLOSE: (--)

Executes the function specified using the PUT.CLOSE.FUNCTION: . More or less a termination of an instrument. CLOSE: is executed when the player containing the instrument finishes. Also FREE:s memory allocated by OPEN:. See below, PUT.OPEN.FUNCTION: , for more information.

DEFAULT: (--)

Defaults are: OFFSET is 0, TUNING is set to 0 (no tuning translation done), GAMUT is set to 0 (same). The ON and OFF functions (interpreters) are set to the values of DEFAULT.ON.INTERP and DEFAULT.OFF.INTERP. These are normally set to INTERP.EL.ON.FOR and 3DROP for simple note playing. The CLOSE and OPEN functionss are set to DROP, the channel range is set to 1 1 and the instrument CHANNEL is set to -1 (no channel currently selected). MUTE is set to FALSE.

DETRANSLATE: (note -- note_index true | false)

Reverses the translation done by the TRANSLATE: method. If the NOTE cannot be generated by the TRANSLATE: method, then a FALSE if returned. This could happen if you passed DETRANSLATE: a note that was "not in the key" determined by the instrument's gamut.

ELEMENT.OFF: (elmnt# shape --)

Exactly like ELEMENT.ON, but executes the Element Off function of the instrument. Usually internal. (see below, ELEMENT.ON:)

ELEMENT.ON: (elmnt# shape -- , process a shapes element)

This method is called by shape players when they are to play a particular element of a shape. The interpreter specified using the PUT.ON.FUNCTION: method will be called when ELEMENT.ON: is called. The ON interpreter must have the following stack diagram:

```
( element# shape instrument --- )
```

FIRST.NOTE.OFF: (-- , turn off last note)

Instruments remember the previous notes you have played using NOTE.ON: . If several notes have been turned on then the "newest" note (you can think of this as a *FIFO* -- first in, first out) will be turned off by this command. You can mix calls to NOTE.OFF: and the FIRST.NOTE.OFF: method. Calls RAW.NOTE.OFF: using Late Binding.

GET.xxx (--- xxx)

There are a number of methods that correspond to PUT.xxx methods. Examples are GET.GAMUT: , GET.TUNING: , etc. See the PUT.xxx: method for explanations.

LAST.NOTE.OFF: (-- , turn off last note)

Instruments remember the previous notes they played using NOTE.ON: . If several notes have been turned on then the "oldest" note (you can think of this as a FIFO -- first in, first out) will be turned off by this command. You can mix calls to NOTE.OFF: and LAST.NOTE.OFF: . Calls RAW.NOTE.OFF: using Late Binding.

NOTE.OFF: (note-index velocity --)

Turns note off in same manner as NOTE.ON: . Deletes the note from the list.

NOTE.ON: (note-index velocity --)

The note index is translated using the TRANSLATE: method and then played by making a late bound call to the RAW.NOTE.ON: method. The note is added to the list so it can be kept track of for ALL.OFF, etc.

NOTE.ON.FOR: (note-index velocity on-time --)

Plays a note for the specified time. Calls NOTE.ON: using the current virtual time.and then increments the virtual time and calls NOTE.OFF:. Both calls use *late binding* so that the NOTE.ON: of the later subclasses is used. Restores virtual time to its original value when done.

OPEN: (--)

Opens an instrument for use. Calls the OPEN function specified using PUT.OPEN.FUNCTION: . If the instrument is already opened, then a count is kept of how many times it has been opened. CLOSE: will decrement that counter until it is 1, then actually close the instrument. Allocates memory to track notes by calling NEW: with the #VOICES set by the PUT.#VOICES: method.

PRINT:

Prints OPEN, ON, OFF, and CLOSE functions (ID's); CHANNEL RANGE, TUNING, GAMUT, and CURRENT-CHANNEL; and NAME of an instrument.

PUT.#VOICES: (#voices --)

Set the maximum number of voices that can be played simultaneously on this instrument. If you call NOTE.ON: several times without calling NOTE.OFF: and exceed this number, then the instrument will start turning off voices. The default is 8 voices. This number determines how much memory will be allocated for note tracking by the OPEN: method.

PUT.CHANNEL: (channel --)

Sets the channel to use in an instrument. This will generally override any channel allocation that might occur. (Remember that "channel" is not always MIDI channel, but can apply also to Amiga Local Sound channels, or anything else the user wishes). See the OB.MIDI.INSTRUMENT class documentation later in this chapter.

PUT.CHANNEL.RANGE: (lo hi -- , sets channel range)

Sets the range of allowable channels for an instrument. Some instrument classes will use this range for automatically allocating a channel for use when an instrument is opened. Note that even though this method will automatically sort the two numbers on the stack, the user should abide by the (LO HI --) convention. It is easy to "force" an instrument to a specific channel by having the arguments to PUT.CHANNEL.RANGE: be the same number. You can also force an instrument to a channel using the PUT.CHANNEL: method.

PUT.CLOSE.FUNCTION: (cfa --)

Puts CFA into the CLOSE.FUNCTION of an instrument. See PUT.OPEN.FUNCTION: .

Note that the stack diagram of a CFA stored in the CLOSE function must be

```
( instrument -- ).
```

PUT.GAMUT: (gamut --)

Puts a predefined gamut, or set of pitches, into an instrument. A gamut is an object of the class TRANSLATOR. Note that gamut is the term used in the instrument, but there is no actual class of objects in HMSL called gamuts. It is used to convert the note on index to a new value. This could be used to restrict notes to, for example, a given "key." If gamut = 0, then there will be no gamut translation by NOTE.ON: . To use the stock gamut, enter:

```
TR-CURRENT-KEY PUT.GAMUT: MY-INSTRUMENT
```

PUT.MUTE: (flag -- , silence output)

Disable the output of an instrument if the flag is true.

PUT.OFF.FUNCTION: (cfa --)

Puts CFA into the off-function of an instrument. Essential to the definition of interpreters. See the description of PUT.ON.FUNCTION: below.

PUT.OFFSET: (offset --)

Puts an OFFSET into the instrument, which is used automatically. This is used by the TRANSLATE: method. The instrument's OFFSET is applied before (!) the gamut translation.

PUT.ON.FUNCTION: (cfa --)

Puts CFA into the on-function of an instrument. Essential in the definition of user-defined custom interpreters.

The stack diagram for the CFA argument to PUT.ON.FUNCTION is:

```
( element# shape instrument -- ).
```

PUT.OPEN.FUNCTION: (cfa --)

Puts CFA of executable routine into OPEN.FUNCTION of an instrument. Note the stack diagram of a CFA stored in the OPEN function must be

```
( instrument -- ).
```

PUT.TUNING: (tuning --)

Puts a predefined tuning in the instrument. This is not used by this generic instrument class but is used by the Amiga Instrument class.

Related Method: GET.TUNING:

RAW.NOTE.OFF: (note velocity --)

Simply print values. Other classes would have much more complex device specific action.

RAW.NOTE.ON: (note velocity --)

Simply print values. Other classes would have much more complex device specific action.

TRANSLATE: (note_index -- actual_note)

This method is used internally by NOTE.ON: and NOTE.OFF: to translate a note index into an actual (real world) note. To generate the new note value, first the note offset is added to the index, then the new note is looked up in the Gamut. The algebraic (not Forth!) equation for this would be:

```
ACTUAL_NOTE = GAMUT ( NOTE_INDEX + OFFSET )
```

OB.MIDI.INSTRUMENT subclass of OB.INSTRUMENT

MIDI instruments are a subclass of instruments defined especially with MIDI in mind. MIDI instruments provide several new features beyond the basic OB.INSTRUMENT class.

One feature is *automatic MIDI channel allocation*. With multitimbral synthesizers, the specific channel used is often unimportant as long as a channel within a certain range is used. If you have a 4 channel multitimbral synthesizer that uses channels 5,6,7 and 8 then you could specify that several MIDI instruments would use any one of those channels by calling the PUT.CHANNEL.RANGE: method. When an instrument is opened, an available channel would be allocated and assigned to that instrument. This way more than 4 instruments could share that synthesizer as long as no more than 4 were open at any one time. In a complex piece it would otherwise be difficult to predict which channels to assign the various instruments. If you want to force an instrument to use a specific channel, then call PUT.CHANNEL: which will turn off the automatic allocation.

MIDI instruments also support *changing presets* .

The definition of MIDI.INSTRUMENTS is intentionally kept quite simple, since we expect users to define more complex ones to meet their own needs. For example, the user might define a class called OB.MY.MIDI.INSTRUMENTS, which is a subclass of OB.MIDI.INSTRUMENTS, and add any instance variables (controllers, system exclusives, etc.) and methods that are needed for a given customized application.

An ALLOCATOR is defined for MIDI, called the MIDI-ALLOCATOR, used to automatically allocate MIDI channels for MIDI INSTRUMENTS. The class OB.ALLOCATOR is described in a later section.

Eight simple MIDI INSTRUMENTS are predefined. They are called INS-MIDI-1 thru INS-MIDI-8.

MIDI INSTRUMENT Methods

Methods are inherited from OB.INSTRUMENT, and will only be documented here where they are new or different. Please see the OB.INSTRUMENT documentation earlier in this chapter.

Method	Stack diagram
CLOSE:	(--)
GET.#VOICES:	(-- #voices, get maximum #voices for polyphony)
GET.PRESET:	(-- preset, returns preset used when instrument opened)
LAST.NOTE.OFF:	(-- , turn off last note)
NOTE.OFF:	(note-index velocity --)
NOTE.ON:	(note-index velocity -- , translate and play)
OPEN:	(-- , opens MIDI instrument)
PRESET:	(preset -- , change MIDI preset if open)
PUT.PRESET:	(preset -- , sets preset for use when instrument opened)

Table 12-2. MIDI Instrument Methods

CLOSE: (--)

Closes MIDI instrument, and deallocates MIDI channel if allocated.

GET.PRESET: (-- preset, returns preset used when instrument opened)

Gets default MIDI preset, which is -1 (no preset, use whatever is currently selected on the synthesizer).

DEFAULT: (--)

Same as OB.INSTRUMENT plus: Default OFFSET is set to 36 (which corresponds to the lowest C on a Casio CZ-101). The preset is set to -1. The channel range is set to 1-16. The #voices is set to 8.

NOTE.OFF: (note-index velocity --)

See below. Does MIDI.NOTEOFF instead of MIDI.NOTEON.

NOTE.ON: (note-index velocity -- , translate and play note)

This is inherited as is from OB.INSTRUMENT which calls RAW.NOTE.ON: using late binding. This method is essential to MIDI instrument interpreters, and useful for testing from the keyboard. It is also useful in writing actions and productions which might want to "bang" a MIDI instrument.

OPEN: (-- , opens MIDI instrument)

Opens instrument for use. If a channel has not been set using PUT.CHANNEL: then a free channel will be assigned to the instrument. The range of channels to choose from is set using the method PUT.CHANNEL.RANGE: . See discussion in the introduction to MIDI instruments above. Finally calls Open function set by PUT.OPEN.FUNCTION: .

PRESET: (preset -- , change MIDI preset)

Changes PRESET on current CHANNEL assigned to the instrument if instrument is open.

PUT.CHANNEL: (channel --)

Sets the MIDI channel to use in an instrument. If a channel is not explicitly set using this method, then a channel will be allocated within the channel range. If you have a limited number of channels, this method is handy for forcing instruments to share channels.

PUT.CHANNEL.RANGE: (lo hi -- , sets channel range)

Sets the range of allowable channels for an instrument. When the instrument is opened, it will use this range for automatically allocating a MIDI channel. Note that even though this method will automatically sort the

two numbers on the stack, the user should abide by the (LO HI --) convention. It is easy to "force" an instrument to a specific channel by having the arguments to PUT.CHANNEL.RANGE: be the same number. You can also force an instrument to a channel using the PUT.CHANNEL: method.

PUT.PRESET: (preset | -1 -- , sets preset for use when instr. opened)

A preset change will be sent over MIDI when the instrument is opened (done when the player that contains it is executed). A preset value of -1 (the default value) indicates that a "change preset" command will not be sent over MIDI, so a user can manually set the preset from the instrument itself.

RAW.NOTE.OFF: (note velocity --)

See below. Does MIDI.NOTEOFF instead of MIDI.NOTEON.

RAW.NOTE.ON: (note velocity -- , play MIDI note)

This is the primary device specific method for this instrument. It simply sets the appropriate MIDI channel and calls MIDI.NOTEON.

OB.ALLOCATOR subclass of OB.BARRAY

Allocators are used to allocate a shared resource like MIDI channels, local sound channels, etc. They are mainly used internally in the channel allocation mechanisms for MIDI instruments, but can be useful in customizing instrument definitions and so are documented briefly here. Allocators may be considered as a simple array of checked or unchecked cells, with methods for checking (MARK:) , unchecking (DEALLOCATE:), or returning the next unchecked cell (ALLOCATE:).

Allocators are a subclass of OB.BARRAY.

Allocator Methods

Method	Stack diagram
ALLOCATE:	(-- index true | false, allocate one if available)
ALLOCATE.BLOCK:	(#-desired -- index true | false)
ALLOCATE.BLOCK.RANGE:	(#-desired lo hi -- index true | false)
ALLOCATE.RANGE:	(lo hi -- index true | false, allocate within range)
CLEAR:	(-- , sets everything to deallocated state)
DEALLOCATE:	(index -- , deallocates that index)
DEALLOCATE.BLOCK:	(index count --)
GET.OFFSET:	(-- offset)
MARK:	(index -- , mark as allocated whether free or not)
NEW:	(--)
PUT.OFFSET:	(offset --)

ALLOCATE: (-- index true | false, allocate one if available)

Most useful and high level method for this class, uses lower-level methods to find an index within range. Returns index and true; or false.

ALLOCATE.BLOCK: (#-desired -- index true | false)

Gives a contiguous block of resources. Used by HMSL for generating modulating pairs of channels for Amiga Instruments, but may be used for any other similar situation.

ALLOCATE.BLOCK.RANGE: (#-desired lo hi -- index true | false)

Same as ALLOCATE.BLOCK: , but restricts range.

ALLOCATE.RANGE: (lo hi -- index true | false, allocate within range)

Allocates a value within range, and gives a true flag, or just returns a false flag if it could not allocate a value within the indices.

CLEAR: (-- , make everything available)

This method will deallocate everything and is used to initialize an allocator.

DEALLOCATE: (index --)

Deallocates indexed resource.

DEALLOCATE.BLOCK: (index count --)

DEALLOCATE:s count number of resources starting at index. Usually paired with ALLOCATE.BLOCK: .

GET.OFFSET: (-- offset)

Returns allocator's offset.

INIT: (--)

Default offset is 0.

MARK: (index -- , mark as allocated)

This is useful for setting a particular index as already allocated, whether it is free or not. This can be used to reserve a resource or when something must use a specific resource, and not just any one from the total pool of available resources.

NEW: (--)

Like all NEW:'s, but also clears the allocator.

PUT.OFFSET: (offset --)

Puts offset into an allocator. The offset is used in conjunction with the parameter returned by ALLOCATE: (see above) to indicate which cell in the range is being referred to. For example, a typical MIDI allocator contains slots for channels 1-16, but they are referred to internally as slots 0-15. Thus the offset for such an allocator is 1; allocating channel 1 is equivalent to marking slot 0 as taken.

Interpreters

What Happens When an Element is Played

When a player decides that the next element of a shape is to be played, it passes the element number and the shape's address to the instrument using the **ELEMENT.ON:** method. The instrument then puts its address on the stack and calls the *on interpreter*. The interpreter can do anything it wants with the information. The default interpreter extracts a note index and a velocity from the shape and calls **NOTE.ON.FOR:** for the instrument on the stack. In **NOTE.ON:** , the instrument converts the note index to a note value (eg. MIDI note value) using the *offset* and *gamut* and plays the note. If **PLAY.ON&OFF:** has been called for the player then it will send an **ELEMENT.OFF:** message to the instrument after the *on time* has elapsed.

The player is responsible for determining the *on time* of an element. This is generally only relevant for notes that need a MIDI Note On message and a MIDI Note Off message. Remember that for notes:

On Time = Time between Note On and Off for a given note.

Duration = Time between Note Ons for two successive notes.

When a player determines the *on time for an element* it sets the value ON.TIME which can be read by interpreters. This is handy if you want to call the NOTE.ON.FOR: method.

ON.TIME (-- on-time)

Writing Interpreters

It is **critical** that interpreters conform strictly to the following stack diagram. They are passed the *ELEMENT#* and the SHAPE that is being played by the player, plus the address of the instrument. The stack diagram for both *ON* and *OFF* interpreters is therefore:

any.interpreter (element# shape instrument --)

Any word that conforms to this stack diagram can be used as an interpreter. An interpreter would typically extract data from the shape using late bound calls to GET: or ED.AT: , process the data in some way, then use some low level methods of the instrument to output that data. NOTE.ON.FOR: and NOTE.OFF: are often used to output the shape data as notes. Here is a handy word that can be used inside interpreters to get the note and velocity from the shape.

INTERP.EXTRACT.PV (element# shape -- pitch velocity)

A utility used inside the standard interpreters. Assumes that dimension 1 is pitch and dimension 2 is velocity. If the shape contains no velocity dimension this word uses a constant velocity value of 64.

Specifying Interpreters

Interpreters are specified for an instrument using the **PUT.ON.FUNCTION:** and the **PUT.OFF.FUNCTION:** methods. If, for example, you had a pair of interpreters named MY-INTERPRETER.ON and MY-INTERPRETER.OFF then you would use the following syntax:

```
'C MY-INTERPRETER.ON  PUT.ON.FUNCTION:  MY-INSTRUMENT
'C MY-INTERPRETER.OFF PUT.OFF.FUNCTION: MY-INSTRUMENT
```

In addition, an OPEN and CLOSE function are sometimes required when an instrument is used. They are called when a player or a job opens or closes an instrument for use. They are placed in an instrument in a way similar to interpreters:

```
'C MY-CLOSE.FUNCTION PUT.CLOSE.FUNCTION: MY-INSTRUMENT
'C MY-OPEN.FUNCTION  PUT.OPEN.FUNCTION:  MY-INSTRUMENT
```

The OPEN and CLOSE functions are passed the address of the instrument only:

open.or.close.function (instrument --)

Predefined Interpreters

A number of Interpreters have been predefined for ease of use. Please see the file **H:INTERPRETERS** to see how these are defined. They are not intended to be the final word on interpreters. If these interpreters do something close to what you want, but not exactly, then write new ones based on these. We encourage people to write new interpreters because it is one of the more powerful features of HMSL.

INTERP.EL.OFF (element# shape instr --)

The same as INTERP.EL.ON except NOTE.OFF: is called.

INTERP.EL.ON (element# shape instr --)

Treats dimension 1 and 2 as note and velocity respectively. Calls NOTE.ON: for the instrument passed. A note value of 0 is assumed to be a rest. A velocity of 0 will turn a note off that is already sounding. This word uses INTERP.EXTRACT.PV to get the data from the shape. Since this does not use NOTE.ON.FOR: you must turn the note off using INTERP.EL.OFF or INTERP.FIRST.OFF or INTERP.LAST.OFF or have a separate note off event with a zero velocity. This interpreter is often used for playing back recorded shapes that are in "expanded form", ie. separate note ON and OFF events.

INTERP.EL.ON.FOR (element# shape instr --)

This is the **default** interpreter. It treats dimension 1 and 2 as note and velocity respectively. Calls NOTE.ON.FOR: for the instrument passed. A note value of 0 is assumed to be a rest. The on-time for the note is determined by calling the value ON.TIME which is set by the player.

INTERP.EXECUTE (element# shape instr --)

This interpreter allows you to schedule the execution of Forth functions. It allows you to mix different types of activity very freely. The only limitation is that all of the functions referenced in any given shape *must* have the same general stack diagram. This means they must all remove the *same* number of items from the stack and leave *none*. If your functions remove N items then the shape must have N+2 dimensions. The zero dimension can be anything, but will probably be durations. The highest numbered dimension must contains the *CFA*'s of the functions. The dimensions contain the data parameters to the functions which can be anything you want. Different shapes with different functions and different numbers of dimensions can be used simultaneously as long as they are internally consistent.

The definition of this interpreter is very simple and instructive:

```
: INTERP.EXECUTE  ( element# shape instr -- )
   DROP  ( don't need instrument )
   GET:  []   ( -- time data_1 ... data_N cfa )
   EXECUTE  DROP  ( do it then drop time )
;
```

The stack diagram for the function must be:

```
your.function  ( data_1 ... data_N -- )
```

The interpreter will pass N values from the shape to your function, which *must* get eaten. As an example, consider a shape that calls MIDI.NOTEON and MIDI.CONTROL. These both take two data items on the stack. The shape must therefore have 2+2=4 dimensions. Here is how you would setup one of these shapes.

```
40 4 NEW: MY-SHAPE
20  45   80  'C MIDI.NOTEON  ADD: MY-SHAPE
10   1   57  'C MIDI.CONTROL ADD: MY-SHAPE
15  45    0  'C MIDI.NOTEOFF ADD: MY-SHAPE
```

Any other Forth word that took two items from the stack and returned nothing, e.g. 2DROP, GR.DRAW, etc., could be used.

Use PRINT.EXEC.SHAPE to print one of these kinds of shapes.

INTERP.LAST.OFF (element# shape instr --)

Calls the method LAST.NOTE.OFF: for the instrument, and disregards SHAPE and ELEMENT#. Useful if the shape is being modified in real-time (for example, if it changes between NOTE.ON: and NOTE.OFF:), or if the ON interpreter is generating values that you cannot recreate or read in some way, for example, random values. If INTERP.EL.OFF was used in these cases then notes would be left hanging.

INTERP.PLAY.MORPH (element# shape instr --)

This interpreter allows you to schedule the playing of morphs, eg. players, collections, etc. at arbitrary times. The shape is assumed to contain a morph address in dimension 1, and a repeat count in dimension 2. The morph will be stopped after ON.TIME has elapsed unless ON.TIME is zero. To force the ON.TIME to zero use a duty cycle of (0,1). See the file HP:SCHEDULE_MORPHS for an example of its use. Use PRINT.MORPH.SHAPE to print one of these kinds of shapes.

Interpreter Example

To illustrate the use of another custom interpreter, let's define an interpreter that uses dimension 1 as relative pitch, or interval. This example uses the return stack as a place to store data instead of using local variables. This example uses an OPEN function to reset the note each time it opens. Otherwise it could drift out of range.

```
: REL.OPEN  ( instrument -- , reset last-note )
   40 SWAP PUT.DATA: []  ( use data slot for last note )
;
: REL.ON  ( element# shape instrument -- , play relative note )
```

```
   >R ( -- e s , save for later )
   INTERP.EXTRACT.PV    ( -- offset v , get dim 1 and 2 values )
   SWAP R@ GET.DATA: [] ( -- v offset old-note , from instrument )
   +   ( -- v new-note , calc relative note )
   DUP R@ PUT.DATA: []      ( save for next time )
   SWAP ON.TIME R> NOTE.ON: []  ( play on instrument )
;
: USE.RELATIVE.INTERPRETER  ( instrument --)
   'C REL.OPEN OVER PUT.OPEN.FUNCTION: []
   'C REL.ON SWAP PUT.ON.FUNCTION: []
;
MY-INSTRUMENT USE.RELATIVE.INTERPRETER
```

Note the simple Forth technique in the previous word, USE.RELATIVE.INTERPRETER of using the Forth stack manipulation OVER to pass the instrument address to the various methods. This technique can be very useful in HMSL in general, since several of the objects pass their own addresses to their methods and to other objects in this way (like instruments and jobs).

Another way to accomplish the same thing would be to use local variables. For example, this word can be rewritten, with no stack manipulation, as:

```
: USE.RELATIVE.INTERPRETER  { instrument --}
   'C REL.OPEN INSTRUMENT PUT.OPEN.FUNCTION: []
   'C REL.ON INSTRUMENT PUT.ON.FUNCTION: []
;
```

This is in fact how most HMSL programmers generally define interpreters and other functions. For example, try rewriting the above routine **REL.ON** by replacing the first line of code with local variables:

```
: REL.ON  { element# shape instrument -- , play relative note }
```

Note that all the Forth stack manipulation words (SWAP, R@, DUP, >R, etc.) will dissapear.