Chapter 11
Translators

Most Important Information

Translators are used in HMSL for translating indices to notes, notes to pitches, etc. They are what allow you to restrict output to specific keys. The most important OB.TRANSLATOR methods are **TRANSLATE:** , **TO:** , **STUFF:** , **PUT.TRANSLATE.FUNCTION:** , **PUT.MODULUS:** and **PUT.OFFSET:** .The most important OB.TUNING.RATIOS method is **PUT.1/1:** .

Introduction

Translators are a class of HMSL objects which *convert from one numeric system to another.* Examples might include converting a generic note index to a MIDI value in a specific set of pitches (i.e., gamuts in instruments), or converting a note index to a pitch or period value for an instrument with a given tuning.

OB.TRANSLATOR is a subclass of OB.ARRAY, with additional instance variables for offset and modulus.

Use TO: or STUFF: to put values in the translators. AT: will retrieve values, as in OB.ARRAYs.

Translators are extremely useful and powerful for adding intelligence to instruments. Translators may also be used by productions, actions, and behaviors. Generally, if you want to simply accept one value and return another, translators are the best thing to use.

NOTE.ON: is the only method currently in use that automatically calls the translator for that instrument (if there is one). The translator is put in the instrument via the PUT.TUNING: or PUT.GAMUT: method for instruments. Tunings and gamuts are specialized translators.

Definition of selected terms:

Translate To convert from an input value to an output value.

Detranslate To perform the inverse function of translate.

Offset A number added to every value before being output.

Modulus The length of a repeating pattern in the output (usually for table-driven translators). For conventional western scales this might be 12, for 12 notes per octave.

Size The number of entries in the translation table.

Table-Driven and Function-Driven Translators

There are two ways in which translators may be used: they may be table-driven, or function-driven. In table-driven translators, the translation formula divides the input index by the size of the translation table, giving a quotient and remainder. The quotient gets multiplied by the modulus. The remainder indexes into the translation table. The resulting two values and the offset are all added together and returned. This technique is what allows you to specify notes for just one octave and then have the pattern repeated automatically for other octaves. For a twelve note per octave system, the modulus would be twelve.

The formula is:

```
VALUE = ((INDEX/LENGTH)*MODULUS) + TABLE[REM(INDEX,LENGTH)]+OFFSET
```

Translators are table-driven by default. If a CFA (non-zero) is stored in a translator via the PUT.TRANSLATE.FUNCTION: , the translator will not use its table (or offset or modulus), but will instead use the user-defined function, which must have the following stack diagram:

```
( input translator --- output )
```

Since the address of the translator itself is passed to the function, the function could use the data in the table, the offset, or the modulus in some arbitrary way.

Translator Methods

OB.TRANSLATOR inherits the methods of OB.ARRAY and it has the following additional methods:

Method	Stack diagram
DETRANSLATE:	(value -- index | flag , reverse translate)
GET.MODULUS:	(-- modulus, fetches repeat length)
GET.OFFSET:	(-- offset , fetches offset)
GET.TRANSLATE.FUNCTION:	(-- cfa , get custom translator)
PUT.MODULUS:	(modulus -- , store repeat length)
PUT.OFFSET:	(offset -- , store value to add at end)
PUT.TRANSLATE.FUNCTION:	(cfa -- , set custom translator)
TO:	(value index -- , set item in translator array)
TRANSLATE:	(index -- value , convert index to value)

Translator Methods

DETRANSLATE: (value -- index | flag , reverse translate)

Attempts to reverse the translation. Returns the index from which the value would have been derived, or returns a flag of 0 to indicate that the output would be invalid. This will not work if there is a translate function.

Related Word: TRANSLATE:

GET.MODULUS: (-- modulus , fetches repeat length)

Related word: PUT.MODULUS:

GET.OFFSET: (-- offset , fetches offset)

Related word: PUT.OFFSET:

GET.TRANSLATE.FUNCTION: (-- cfa , get custom translator)

Returns CFA of custom translator function. If zero, the translator is a "table-driven" translator.

INIT: (-- , initializes)

Sets the offset to zero and the modulus to 12. Note that as usual, INIT: is done when a translator is created, and is rarely used directly.

PUT.MODULUS: (modulus -- , store repeat length)

Stores modulus value, the value to add when the translation table overflows the original data and repeats its pattern. Default is 12. Related word: GET.MODULUS

PUT.OFFSET: (offset -- , store value to add at end)

Stores offset value.

When defining new translators (or, for example, if you're altering one of the simple demos), make sure you are aware of the way that the offset of the translator relates to the values that the indices are expected to produce. One common result of an offset that is too high, for example, will be clipping of values.

Related word: GET.OFFSET:

PUT.TRANSLATE.FUNCTION: (cfa -- , set custom translator)

Specifies what function to use for translation. A CFA of 0 indicates that the translator will be table-driven.

TRANSLATE: (index -- value , convert index to value)

Generates the proper value based on the value. If there is no TRANSLATE.FUNCTION, it will simply look up the value in its internal table.

Some Translator Examples

Using a Simple Translator

Let's define a translator that converts indices to the Fibonacci sequence. The Fibonacci sequence is 1,1,2,3,5,8,13,21, etc. . Let's suppose that we want each octave to use just the first six values. In successive octaves we will simply add 12 to the original sequence. This gives us an "octave replicating" pattern like the minor and major scales.

```
OB.TRANSLATOR TR-1    ( make a translator )
6 NEW: TR-1    ( make room for 6 values )
8 5 3 2 1 1    6 STUFF: TR-1   ( fill translator , not 0STUFF: !!)
PRINT: TR-1    ( to see pattern )

: SHOW.TR-1   ( N -- , show pattern of translation )
   0 DO
      I DUP . TRANSLATE: TR-1 . CR
   LOOP
;

16 SHOW.TR-1   ( first 15 translated values )
( This excerpted table shows how the modulus effects the result. )
4  5     ( input values within original range 0-5 )
5  8
6  13    ( 13 = 1 + 12 , modulus added once for each time around )
7  13    ( 13 = 1 + 12 )
8  14    ( 14 = 2 + 12 )
9  15    ( 15 = 3 + 12 )
10 17    ( 17 = 5 + 12 )

20 PUT.OFFSET: TR-1
16 SHOW.TR-1      ( all results increased by 20 )
```

Creating a Custom Function

The following is a very simple example of creating a translator, defining a simple TRANSLATOR-FUNCTION, and testing that translator. This translator would most likely be used inside an instrument (by **PUT.TUNING:** or **PUT.GAMUT:** , which would then be used by **NOTE.ON:** in the instrument's interpreter), but it could also be used by a production, action, etc.

First, define the translator and a simple custom TRANSLATE function:

```
OB.TRANSLATOR MY-TRANSLATOR
32 PUT.MODULUS: MY-TRANSLATOR
: MY-TRANSLATOR.FUNCTION  ( input translator -- output )
   GET.MODULUS: [] ( gets current modulus )
   > IF 1 ( bigger than the mod, 1 on stack )
   ELSE 0 ( if not, zero )
   THEN
;
( put the CFA in the translator )
'C MY-TRANSLATOR.FUNCTION
PUT.TRANSLATE.FUNCTION: MY-TRANSLATOR
```

Now to try this out:

```
16 TRANSLATE: MY-TRANSLATOR .
```

```
   ( should return 0 )
50 TRANSLATE: MY-TRANSLATOR .
   ( should return 1 )
```

Stock Translator

HMSL contains a simple translator called TR-CURRENT-KEY, which can do simple translations into conventional musical keys.

TR.SET.KEY (key_OFFSET VN-1 VN-2 ... V0 N -- , sets key)

This word is defined as follows:

```
: TR.SET.KEY
   DUP NEW: TR-CURRENT-KEY ( Makes tr-current-key N big )
   STUFF: TR-CURRENT-KEY
   ( Puts the values on the stack into it )
   PUT.OFFSET: TR-CURRENT-KEY ( and puts in the offset )
;
```

This is a good example of how to set up a translator.

TR.MAJOR.KEY (Key_OFFSET -- , set current key to major key)

Uses TR.SET.KEY to form a major key. This word's definition is:

```
11 9 7 5 4 2 0  7 TR.SET.KEY
```

Related Word: TR.MINOR.KEY

TR.HARMONIC.MINOR (Key_OFFSET -- , sets current to minor key)

Same as TR.MAJOR.KEY, except values are: 11 8 7 5 3 2 0.

TR.INDEX->KEY (note_index -- note_in_key)

Translates the note index provided into the current key. (The note index represents the scale degree that you want to retrieve.) This word returns the corresponding note value by applying the translator TR-CURRENT-KEY to it. The definition is simply:

```
TRANSLATE: TR-CURRENT-KEY
```

TR.MIDI.OFF (index velocity -- , translate and turn note off)

Translate the note index into the current key and turns it off with a MIDI.NOTEOFF. Requires that you know the index and the velocity.

Related Word: TR.MIDI.ON

TR.MIDI.ON (index velocity -- , translate and turn note on)

Translates the note index into the current key and turns it on with the specified velocity using MIDI.NOTEON.

Related Word: TR.MIDI.OFF

Examples using Stock Translator

Here is an example of how to use TR-CURRENT-KEY to force an instrument to play in G major:

```
TR_KEY_G  TR.MAJOR.KEY
TR-CURRENT-KEY PUT.GAMUT: INS-MIDI-1
```

Here is an example of using this stock translator to generate a random melody in F# minor:

```
TR_KEY_F# TR.HARMONIC.MINOR  ( Set to F# minor. )
: JAM.F#.MINOR  ( -- , random notes in F# minor )
   BEGIN
      30 CHOOSE ( random note )
      60 CHOOSE 60 + ( and random velocity )
      TR.MIDI.ON
```

```
            200 MSEC  MIDI.LASTOFF
            ?TERMINAL
          UNTIL
      ;
```

Tunings

Introduction

Objects of the class of OB.TUNING contain the frequencies or periods to use for specific notes. There is only one new method, **PLAY:** , the rest of the methods discussed are alterations of methods inherited from OB.TRANSLATOR, of which OB.TUNING is a subclass.

Tunings differ from the parent class translators in that in general they do not use the modulus or the offset. They are intended, rather, as absolute definitions of a set of pitches.

Currently, OB.TUNING is primarily useful on the Amiga, where local sound is implemented (local sound is not currently included in the Macintosh version of HMSL). However, the user might find that the OB.TUNING class provides useful ways performing other tasks (like proportionally related graphics, or formal musical ideas which involve ratiometric ideas), or in the case of MIDI, an HMSL instrument can be designed which, for example, uses the tuning and ratio classes in HMSL to control a tunable MIDI instrument (like the Yamaha TX81Z, FBO1, or DX7 Mark II).

Method	Stack diagram
DETRANSLATE:	(period -- index true | false , reverse translation)
PLAY:	(-- , plays scale)
TRANSLATE:	(index -- period , generate period for a given note)

Table 13-2. Tuning Methods

DETRANSLATE: (period -- index true | false , reverse translation)

Attempts to reverse-translate the given period. If it cannot do so, it issues a flag of 0. If it can reverse-translate, it returns both a true flag and the appropriate index.

PLAY: (-- , plays scale)

Plays tuning once. ?TERMINAL gets you out, waits 1/2 second between each note.

TRANSLATE: (index -- period , generate period for a given note)

Translates the given index into a period, clipping at the tuning limit if necessary. This is one of the main advantages of a tuning, for if you want to have an absolute pitch set definition, you may not want the concept of modulus. You may want, for example, non-octave replicating tunings. Since a tuning can be as big as you please, just define a complex multi-octave tuning, and index into it.

Related Method: DETRANSLATE:

A predefined translator is available, called TUNING-EQUAL. The associated words are to initialize and free it. It is a quarter tone scale, mainly useful for the Amiga Local Sound, with 24 notes in one octave. The word that sets up this tuning is called TU.BUILD.EQUAL.

Tuning Ratios

Class OB.TUNING.RATIOS is a subclass of OB.ELMNTS. This class is used for ratio-based tuning systems. Just-intoned scales can be created using this class.

Objects of this type are 2 dimensional arrays (hence OB.ELMNTS as the superclass, rather than OB.TUNING). The first dimension holds numerators, and the second holds denominators.

In general, use ADD: or PUT: to place ratios in TUNING.RATIOS. Always remember that each place in a TUNING.RATIO consists of two values. Thus, 5 4 ADD: MY-RATIO is the proper syntax.

TUNING.RATIOS store a value for the fundamental, called 1/1, and have two new methods for making use of that fundamental: PUT.1/1: , and GET.1/1: .

Method	Stack diagram
GET.1/1:	(-- 1/1_period -- , base period for 1/1 ratio)
NEW:	(number_of_pairs --)
PUT.1/1:	(1/1_period -- , sets base period for 1/1 ratio)
TRANSLATE:	(index -- period , converts index to period)

Table 13-3. Tuning Ratio Methods

GET.1/1: (-- 1/1_period -- , base period for 1/1 ratio)

Returns absolute value for 1/1, or fundamental of TUNING.RATIO.

Related Word: PUT.1/1:

INIT: (--)

Initializes tuning; sets 1/1 to 9240 (again, this value is only relevant to Amiga Local Sound where it specifies a period). We chose 9240 because it is the result of multiplying together some numbers commonly used as denominators.

NEW: (number_of_pairs --)

Allocates space in the TUNING.RATIO for the specified number of pairs of values (numerator/denominator).

PUT.1/1: (1/1_period -- , sets base period for 1/1 ratio)

Stores fundamental for TUNING.RATIO.

Related word: GET.1/1:

TRANSLATE: (index -- period , converts index to period)

Calculates base period and shifts to proper octave. This means that in general TUNING.RATIOS can be used as octave replicating scales, since their TRANSLATE: method uses a simple algorithm for finding the proper octave. If the index sent to a TUNING.RATIO is greater than the size of the array, it will shift the octave of the resultant period by the appropriate number — that is, if the index is greater than, say, 3 times the number of values, the resultant period will be in three octaves plus the ratio times the fundamental.

Note that this algorithm allows for non-octave-replicating scales, in that the values in a TUNING.RATIOS may span more than one octave. As long as the index into the translator does not exceed the number of ratios stored, then no octave conversion will take place. For example, you can have 40 ratios that span 3 octaves in a non replicating fashion. If, however, you use an index of 45, then you will get the value for 5 transposed up one octave.

Example of Tuning Ratios

Create a simple just intoned scale and use it in an Amiga Instrument.(Note that it builds the scale downwards — it just depends on how you ADD: the ratios to the object).

```
OB.TUNING.RATIOS  MY-RATIOS
4 NEW: MY-RATIOS
1 1 ADD: MY-RATIOS ( fundamental )
2 3 ADD: MY-RATIOS ( perfect fifth )
4 5 ADD: MY-RATIOS ( major third )
6 7 ADD: MY-RATIOS ( septimal minor third )
MY-RATIOS PUT.TUNING: INS-AMIGA-1
```

Stock Tuning Ratios

RATIOS-SLENDRO is a five note scale loosely based on tunings of central Javanese (Surakarta) gamelan (1/1, 8/7, 64/49, 3/2, 12/7).

RATIOS-OVERTONE is a just tuning based on the superparticular ratios of the 12-24 members of the harmonic series. These tunings are included for example purposes only. RATIOS-OVERTONE is created by the following code:

```
: BUILD.RATIOS-OVETONE
   12 NEW: RATIOS-OVERTONE
   12 0 DO
      I 12 + 12 ADD: RATIOS-OVERTONE
   LOOP
;
```

Some Ratio Approximations for Equal Temperaments

The following ratios may prove useful to those wishing to experiment with equal divisions of the octave between 2 and 30. These ratios will give a "pretty good" interval for adjacent pitches in those temperaments (for example, for the number 12 the interval 196/185 is "close" to the ratio: ((12th root of 2):1), which is the exact ratio for the 12-tone equal tempered half-step). These ratios are approximations, at best, and error will also be involved as a result of the fixed point math and the inherent frequency accuracy of whatever sound driver you are using (e.g Amiga local sound). However, they should provide a "quick and dirty" way to quickly test out varying equal-temperaments. They are taken from John Chalmers' tables for 1200-tone equal temperament, published in the now out-of-print *Xenharmonikon #1*.

The list is in the form: (# of divisions per octave) numerator/denominator

```
(2) 99/70    (3) 63/50          (4) 44/37
(5) 1024/891   (6) 55/49        (7) 243/220
(8) 12/11    (9) 27/25          (10) 15/14
(11) 16/15   (12) 196/185       (13) 135/128
(14) 21/20   (15) 22/21         (16) 2673/2560
(17) 126/121   (18) 80/77       (19) 28/27
(20) 517/500   (21) 516/500     (22) 1031/1000
(23) 103/100   (24) 1029/1000 (25) 257/255
(26) 1027/1000 (27) 513/500    (28) 41/40
(28) 256/255   (29) 1023/1000 (30) 511/500
```