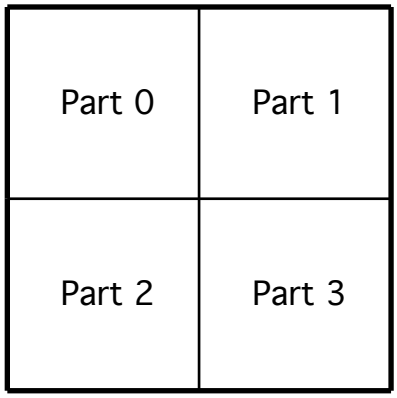
Chapter 15
Interactive Controls

Controls are objects that give you *interactive control* over HMSL using the mouse. Controls are grouped together and placed in a *Screen*. The Shape Editor Screen and Perform Screen were both created using the HMSL Control objects. You can use Controls to *design your own screens* for patch editors, interactive pieces, on screen mixers, etc.

HMSL Controls are built from the graphics primitives of the host computer: draw line, draw text, mouse up, mouse down. Since they do not use the (Host Specific) User Interface tools provided they can be *ported* between different computers running HMSL and still work. The source code for the Shape Editor and the Perform Screen is identical on both the Macintosh and the Amiga. Thus you can design a screen and share it with people who have other kinds of computers.

Control Classes Overview

Lets take a brief look at the different classes of Control Grids. We will study these in more detail later. The most commonly used are the **Menu**, **Check** and **Radio Grids**. They provide a grid of *buttons* that you can click on. Each button is referred to as a *part.* Parts are numbered starting at 0.

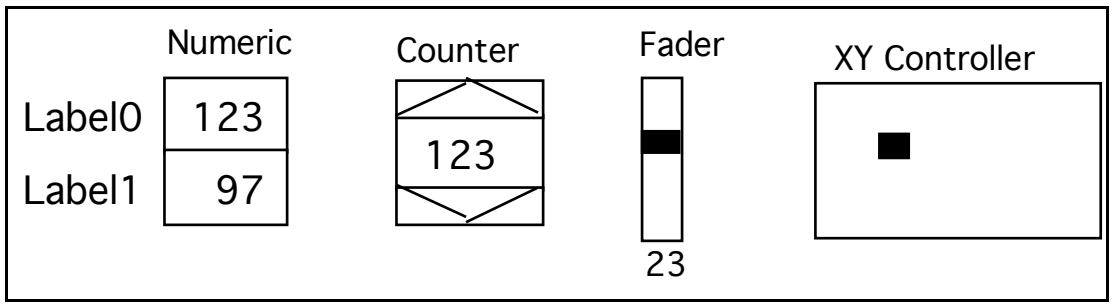


Each button has Text on it and can be turned on or off. The difference between them is in how they turn on and off.

Menu Grids will turn on when you click down with the mouse, and off when you click up. This is handy for triggering commands, or for simulating a synthesizer keyboard.

Check Grids turn on when you click down and stay on until you click down again. This "toggle" action is useful for selecting options, or, for example, in turning on and off a continuous sound or process.

Radio Grids are like the buttons on a car radio. When you click down on one it turns off any others that are already on. This is useful for selecting a mode, or choosing one of a group. The *Mode Selector* in the Shape Editor is a Radio grid.



Counters can be used to select a value. Examples are the *Dimension Selector* in the Shape Editor and the *Priority Selectors* in the Perform Screen. You can also use the Counter to scroll through a list of names by loading it with a special function. An example is the *Shape Selector* in the Shape Editor.

Numeric Grids provide a table of numbers that can be changed by clicking on a number and moving the mouse. When you move the mouse up the number increases and vice versa. Moving the mouse to the right gives coarse adjustment and to the left fine adjustment.

Faders provide a *virtual slidepot* like the faders on a mixer.

For each Control object you can specify a number of properties, size, x,y position, titles, side labels, minimum and maximum values, associated functions, and others.

XY Controllers provide a 2 dimensional controller like a joy stick. You could assign volume to one dimension and pitch to another to create a virtual theremin. You can also disable one of the dimensions to make a vertical or horizontal fader.

Text Grids provide for text input in a grid. This can be used to enter names, or numbers. The mouse is used to select a grid for input and can drag select text. This is essentially a grid of small text editors. You can install filters that can disallow certain characters so that, for example, only numbers could be enterred.

The actual names of these Control Classes are:

OB.CONTROL — used to build other control classes

OB.MENU.GRID — for triggering

OB.CHECK.GRID — toggle for selecting options

OB.RADIO.GRID — for selecting modes

OB.COUNTER — for clicking through values or names

OB.NUMERIC.GRID — for editing a table of numbers

OB.FADER — like the slidepots on a mixer

OB.XY.CONTROLLER — like a joystick

OB.TEXT.GRID — for entering text, including numbers

Control Mouse Functions

You connect the Control to your program by specifying a *function* for it to call when touched. You can specify a *separate function* for mouse button **DOWN**, mouse **MOVE**, and mouse button **UP**.

These functions can do anything you want as long as they have the following stack diagram (again, it can be a good idea to use Local Variables here):

```
MYFUNC  ( value part# -- , do whatever )
```

Most Controls have more than one *part*. In the **Check Grids**, for example, each *small box* is considered a part. Parts are numbered starting from zero. You might use the part number with a CASE statement or as an index into an array.

In a **Check Grid**, each part has a value of either TRUE (-1) or FALSE (0).

You specify which function the control is to call using one of the following methods.

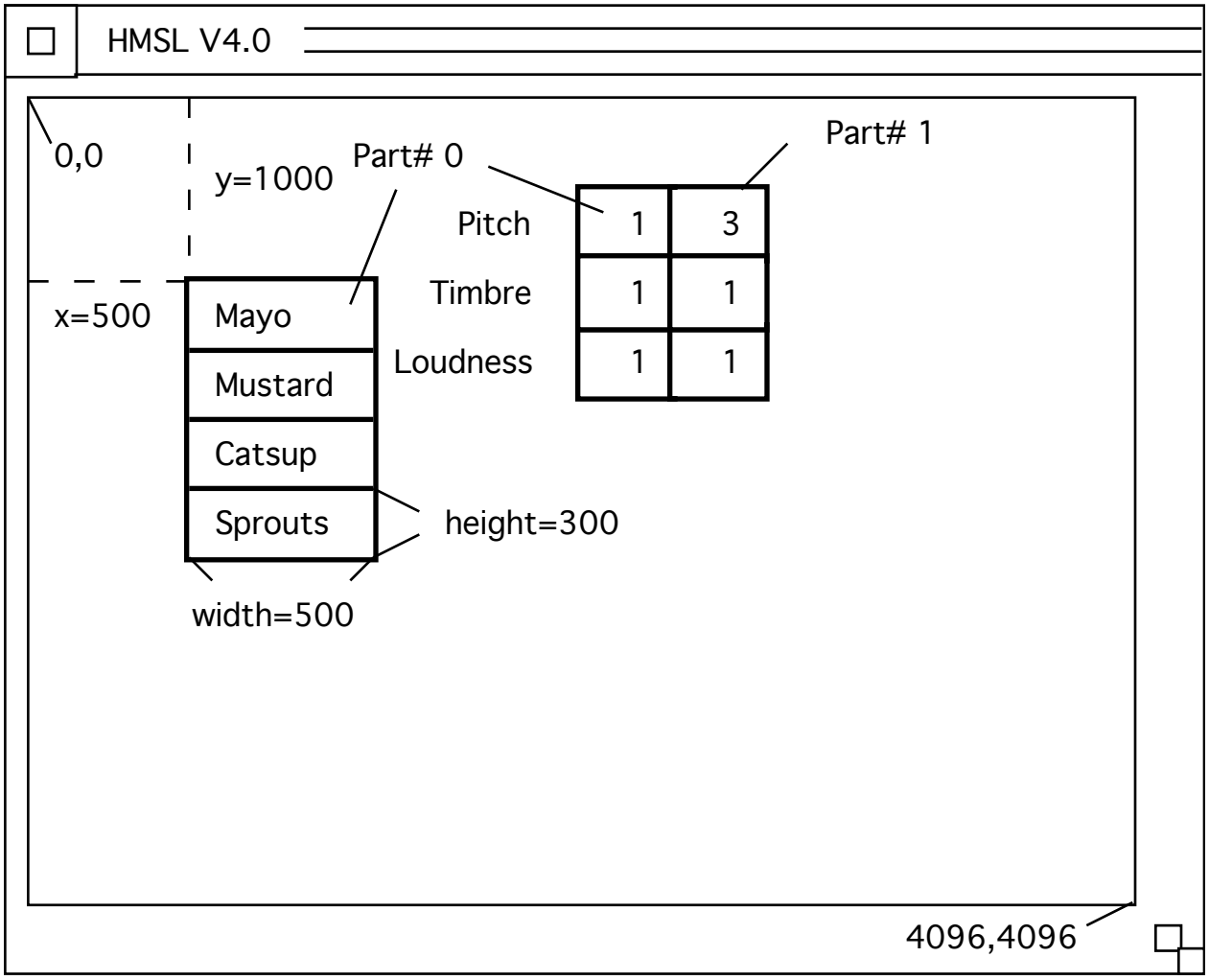
```
PUT.DOWN.FUNCTION:  ( cfa -- )
PUT.MOVE.FUNCTION:  ( cfa -- )
PUT.UP.FUNCTION:  ( cfa -- )
```

There are also corresponding GET... methods that return the CFA.

Control Size and Placement

The size and placement of controls is specified in a *virtual*, or "world" coordinate system that uses a *4096 by 4096* rectangle. The *0,0* point is in the *upper left hand corner* and the *4096,4096* point is in the *lower right hand corner* of the window. Using this system gives us freedom from the pixel resolution of the host computer. If you have room on your computer screen, you can use values larger than 4096 but you will probably have to make the window larger to see them. The Amiga and Macintosh have different resolutions but this is transparent to the user when designing grids.

Because the "world" is square, and most computer screens are rectangular, the "aspect ratio" of controls can vary from one machine to another. [Technical note: Controls use the SCG window and viewport #0. Look in the file h:scg for ways to change the size of the viewport if desired.]



World Coordinate System of Screen
Figure 15.2

When you place a Control in a screen, you specify the *x,y location* of the *top left corner* of the Control. This is done when placing the Control in a *Screen*. You also specify the *width and height*. These values are local to the Control itself. For Controls that use a grid, like a Check Grid, the width and height is the width and height of a *single part*.

Let's look at an example of designing a Control to see how all this works.

Tutorial 1 - Check Grid

Each tutorial will build on the previous one so you should enter this in a file.

Let's design a simple Check Grid to get a feel for this system. The first step is to instantiate the Control Object. Enter:

```
ANEW TASK-CGTEST  ( mark beginning of file )
OB.CHECK.GRID  MY-CHECK
```

We will now write a function that will help us see how these controls work. Let's write a word that simply prints out the value and part number. (We could have used Local Variables here, but it's so simple that they're not really needed).

```
: CHECK.FUNC  ( value part -- , just print them )
   ." Part = " .
   ." , Value = " . cr
;
```

Now let's write a word that sets up the Control. We usually call these "build" words.

```
: BUILD.MY-CHECK  ( -- )
\ Allocate memory for 4 parts, 1 wide, 4 high.
   1 4 NEW: MY-CHECK   (  )
\
\ Now specify the width,height of each part.
   500 300 PUT.WH: MY-CHECK
\
\ Tell it to call our function
   'C CHECK.FUNC  PUT.DOWN.FUNCTION: MY-CHECK
\
\ Specify the text for each part )
   STUFF{ " Mayo"   " Mustard"  " Catsup"  " Sprouts"
   }STUFF.TEXT: MY-CHECK
;
```

We must now place this Control in a *Screen*. Screens can hold several controls.

```
OB.SCREEN  MY-SCREEN

: MYSC.INIT  ( -- , set up controls and screen )
\ Place Grid in Screen and specify x,y
\ You must allocate 3 dimensions for the screen,
\ Dim0 = control, Dim1,2 = x,y.
   4 3 NEW: MY-SCREEN  ( room for 4 controls )
   BUILD.MY-CHECK  ( set up control )
\ Place top-left corner at x,y = 500,1000
   MY-CHECK 500 1000   ADD: MY-SCREEN
\
   " Test1" PUT.TITLE: MY-SCREEN ( title for menu )
;
```

We must also write a word that will clean up afterwards.

```
: MYSC.TERM  ( -- , clean up )
   FREEALL: MY-SCREEN  ( FREE ANY CONTROLS )
   FREE: MY-SCREEN
;
\ Tell HMSL to automatically cleanup if forgotten.
```

```
IF.FORGOTTEN  MYSC.TERM
```

We can now INCLUDE our file and test the Control. Enter directly (not in the file):

```
MYSC.INIT
PRINT: MY-CHECK   ( is CHECK.FUNC listed? )
PRINT: MY-SCREEN  ( is MY-CHECK in there? )
HMSL
```

Now look in the HMSL "Screens" menu for "Test1" which is our screen's title. Select it and you should see your Control appear. Click on it and you should see a message in the Forth window telling you the *Part#* and *Value*. If you get a stack error, check your CHECK.FUNC by passing a value and part and making sure it eats the two values.

When you are done, quit HMSL, then enter:

```
MYSC.TERM
```

Tutorial 2 - Numeric Grid

In this tutorial we will add a numeric grid to the screen we built in the first tutorial.

First we will write the functions that the grid will use. These functions will again just print to the Forth window. These functions could just as easily output MIDI data or do something else more interesting. The reason we're just printing is that it is relatively foolproof – we're not reliant on MIDI devices and audio connections, and subject to their possible problems. Once you know the grid is working, try to change these functions to do something sonic.

Place this code right after the definition of BUILD.MY-CHECK.

```
\ Numeric Grid
OB.NUMERIC.GRID  MY-NUMERIC
: MN.DOWN  ( VALUE PART# -- , Start display )
   CR ." PART# " . ."  = " .
;
: MN.MOVE&UP ( VALUE PART# -- , Show new value )
   DROP . CR?
;
```

We can specify a minimum and a maximum value for each part of the grid using the PUT.MIN: and PUT.MAX: methods.

```
: BUILD.MY-NUMERIC  ( -- , setup grid )
\ Allocate memory for 2 wide, 3 high grid.
   2 3 NEW: MY-NUMERIC
   200 300 PUT.WH: MY-NUMERIC
\
\ A part number of (-1) means ALL parts.
   1  -1 PUT.MIN: MY-NUMERIC
   10 -1 PUT.MAX: MY-NUMERIC
\
\ Set value to 3 for part 1.
   3 1 PUT.VALUE:  MY-NUMERIC
\
\ Specify which functions to call.
   'C MN.DOWN PUT.DOWN.FUNCTION: MY-NUMERIC
   'C MN.MOVE&UP PUT.MOVE.FUNCTION: MY-NUMERIC
   'C MN.MOVE&UP PUT.UP.FUNCTION: MY-NUMERIC
\
\ Specify special limits for part 5.
```

```
      0   5 PUT.MIN: MY-NUMERIC
      100 5 PUT.MAX: MY-NUMERIC
   \
   \ Specify labels for left edge.
      STUFF{
         " Pitch" " Timbre" " Loudness"
      }STUFF.TEXT: MY-NUMERIC
   ;
```

!!Important: If you don't pass 2 values to the NEW: method of any Control that expects them (like Numerics, Radio, Check, or Menu), serious system crashes may occur, since you have no idea what kind of strange value the Control will use as the second number*!!*

Now edit the word MYSC.INIT. Add the following two lines right after the call to ADD: MY-SCREEN .

```
      BUILD.MY-NUMERIC
      MY-NUMERIC 2000 500 ADD: MY-SCREEN \ places it on screen
                     \ uses coordinates 2000 500 for x, y
      MY-SCREEN DEFAULT-SCREEN !  \ make it draw before shape editor
```

Now let's write a word that will tie all this together. Add this to the end of the file.

```
   : TEST ( -- )
      MYSC.INIT
      HMSL
      MYSC.TERM
   ;
```

Save and INCLUDE the file, then enter directly (not in the file):

```
   TEST
```

Arrange the windows so that you can see the bottom part of the Forth window. Try clicking on the top half of one of the numeric grids. Notice how the value increments. If you click on the lower part, it should decrement. The values will change until you hit the limits.

The *Numeric Grids* support *mouse dragging*. Click on one of the parts and hold down the button. Move the mouse up and down and watch the values change. The MOVE function is being called. When you lift up your finger, the UP function is called.

Now click and drag on the lower right part. This is part number 5. Notice that it has different limits then the others.

More Example on Disk

Look in these files for more examples of using control grids:

HP:SPLORP - uses faders to control a musical process

HSC:DEMO_CONTROLS - demo of most control types

HSC:DEMO_XY - demo of XY controller

HSC:THEREMIN - Amiga only demo of XY controller

HSC:SEQUENCER - sequencer screen

H:SHAPE_EDITOR - complicated example

By studying how the existing controls are defined, you can learn how to define new classes of control. See:

H:CONTROL - Check, Radio and Menu Grids

H:CTRL_NUMERIC - Numeric Grid

H:CTRL_COUNT - Counter

H:CTRL_TEXT - OB.TEXT.GRID

H:SCREEN - defines OB.SCREEN which holds controls.

(Note some of these files may be moved to HSC:)

Control Grids Reference

The root class of Control is OB.CONTROL. All other classes are subclasses of it. Here is an outline of the subclasses:

```
OBJECT
  OB.CONTROL
     OB.COUNTER
     OB.CONTROL.GRID
        OB.NUMERIC.GRID
        OB.CHECK.GRID
           OB.MENU.GRID
           OB.RADIO.GRID
     OB.TEXT.GRID
     OB.XY.CONTROLLER
```

OB.CONTROL subclass of OBJECT

This class is the *root class* of all the controls. These controls have *only one part* so the part# for methods like PUT.VALUE: will be ignored. The part# is used for classes like the OB.NUMERIC.GRID which have many parts.

?HIT: (x y -- flag , TRUE if control hit)

This is used internally by Screens. The x and y are in pixels.

?DRAWN: (-- flag , TRUE if currently drawn)

DRAW: (-- , draw control)

This also executes the DRAW function, which is something you might want done every time the Control is drawn.

INIT: (-- , initialize values)

FREE: (-- , free any memory allocated by NEW:)

GET.DATA: (-- data , user data)

This is any miscellaneous data you want to have associated with a Control.

GET.DOWN.FUNCTION: (-- cfa , returns function for button down)

GET.DRAW.FUNCTION: (-- cfa , returns function called when drawn)

GET.UNDRAW.FUNCTION: (-- cfa , returns function called when undrawn)

GET.LASTHIT: (-- part# , part last hit by mouse)

Returns part# of the grid last hit.

GET.MOVE.FUNCTION: (-- cfa , returns function for mouse move)

GET.RECT: (part# -- x1 y1 x2 y2 , bounding rectangle in pixels)

GET.TEXT.FUNCTION: (-- cfa)

GET.UP.FUNCTION: (-- cfa , returns function for button up)

GET.VALUE: (part# -- value , fetch value for part)

What is the current value for a part of a control. This is useful for functions that use these values to do something.

GET.XY: (-- x y , return position of top left corner)

GET.XY.DC: (-- x y , return top left x,y in device coordinates)

Device coordinates are the actual pixel coordinates.

GET.WH.DC: (-- width height , width and height in pixels)

GET.WH: (-- width height , return width and height)

MANY: (-- #parts , return number of parts)

PUT.DATA: (data -- , specify user data)

You can set this to whatever you want for your own purposes.

PUT.DOWN.FUNCTION: (cfa -- , specify function for button down)

PUT.DRAW.FUNCTION: (cfa -- , specify function to call when drawn)

PUT.INCREMENT: (n -- , amount to increment by)

This is used by OB.NUMERIC.GRID , OB.COUNTER and OB.FADER to determine how much to increment (or decrement) when you click on the control.

PUT.MAX: (maximum part# -- , specify maximum value)

If the part # is -1, then set maximum for all parts. Default maximum is 127, so that if you forget to do this, it will be a reasonable MIDI range.

PUT.MIN: (minimum part# -- , specify minimum value)

If the part # is -1, then set minimum for all parts. Default minimum is 0.

PUT.UNDRAW.FUNCTION: (cfa -- , function to call when undrawn)

When a new screen is drawn, the old screen is "UNdrawn".

PUT.MOVE.FUNCTION: (cfa -- , specify function for mouse move)

PUT.TEXT.FONT: (font -- , specify text font)

WARNING!: Use of this method will limit portability of your code between different computers. Fonts are very specific to the computer being used. On the Macintosh, use an integer. On the Amiga, use the address of a font structure. Here are some common font indices for the Macintosh:

```
0 = default system font
4 = Monaco
```

PUT.TEXT.FUNCTION: (cfa -- , specify text function)

This is used in various ways by different controls. It can be used as a label function by OB.NUMERIC.GRID, or as a selection by OB.COUNTER. The text function MUST have the following stack diagram.

your.text.function (n -- addr count)

A possible text function is N>TEXT which will convert a number to its text equivalent. If the text never changes, you can use a TEXTROM.

```
TEXTROM MY-ROM ," apple" ," orange" ," banana"
```

```
'C MY-ROM PUT.TEXT.FUNCTION: MY-CONTROL
```

MY-ROM will have the required stack diagram. For N equals zero it will return an ADDR and COUNT for "apple". Note that the above word for laying down text is ," (comma-quote) not ." (dot-quote).

," (<string"> -- , compile string into dictionary)

PUT.TEXT.SIZE: (size -- , specify text size)

Specify the text size which usually means the height in pixels. The default size on the Macintosh is 12. If you want a smaller size, try 9. The results will vary with the font being used.

PUT.TITLE: ($title -- , specify title for control)

This title will be drawn above the top, left corner.

PUT.UP.FUNCTION: (cfa -- , specify function for button up)

PUT.VALUE: (value part# -- , set value for part)

If the part# is (-1), then all parts will be set to this value. The value is automatically clipped to the MIN and MAX for that part before being stored.

PUT.XY: (x y -- , specify position of top left corner)

This method will allow you to associate a specific screen position for this Control. Otherwise, you can define this position when you ADD: the control to a Screen. If you specify values when you ADD: the Control, those values will override the values specified using PUT.XY: . If you want to use the values associated with the Control itself, use -1 for the x and y positions in ADD:ing the Control to the screen.

PUT.WH: (w h -- , specify width and height)

Specify the size of each part of the grid.

UNDRAW: (-- , call undraw function)

Internal Methods for OB.CONTROL

MOUSE.DOWN: (x y -- flag , process mouse button down)

This is called by the Screen class for every mouse down. It returns TRUE if hit. If it is hit, the Screen will also call this control for MOUSE.MOVE and MOUSE.UP events. It calls EXEC.DOWN: via late binding. The x and y are stored in variables called CG-FIRST-MX and CG-FIRST-MY .

MOUSE.MOVE: (x y -- , process mouse move)

This calls the MOUSE.MOVE functions. It also sets CG-LAST-MX and CG-LAST-MY to X and Y.

MOUSE.UP: (x y -- , process mouse button UP)

This calls the MOUSE.UP functions. It also sets CG-LAST-MX and CG-LAST-MY to X and Y.

OB.CONTROL.GRID subclass of OB.CONTROL

This class has internal storage for multiple values. Thus part numbers are important when using PUT.VALUE: and GET.VALUE: . It also has a storage area for flags which are currently used for Enable/Disable The control grids are drawn as an N by M grid. The number of x and y grid cells is determined using the NEW: method which also allocates the internal value storage array.

CLEAR.PART: (part -- , clear that parts rectangle)

The control should be drawn before calling this.

FREE: (-- , deallocate any allocated memory, important)

GET.ENABLE: (part# -- flag , is it enabled?)

HIGHLIGHT: (part# -- , highlight a part)

Highlighting is done using the XOR graphics mode. Thus to UNhighlight, just call this method again.

PUT.ENABLE: (flag part# -- , enable or disable a part)

When FLAG=FALSE, the part specified will be disabled. A disabled part will be drawn in a special color, or font, which will indicate that it is disabled. If the user clicks on this part, the system error "beep" will be called. The beep may be audible or just a flash on the screen.

NEW: (numx numy -- , allocate memory, specify layout)

This method will allocate enough memory for NUMX*NUMY values.

OB.CHECK.GRID subclass of OB.CONTROL.GRID

When a part of this class is clicked down on with the mouse, it toggles **On** or **Off**. The On state is marked by highlighting the part. This class adds the ability to specify **text** for each part of the grid. The text indicates the function of that part of the grid. The text will be drawn offset from the top left of the part. If a *Text Function* is specified (see the methods for OB.CONTROL) using **PUT.TEXT.FUNCTION:** then that will override the Text specified using **PUT.TEXT:** or **}STUFF.TEXT:**.

}STUFF.TEXT: (stuff{ $0 $1 $2 .. $n-1 -- , stuff text)

This is used to initialize the text for a control. You must call STUFF{ before listing your text strings. For example:

```
STUFF{  " Bong"  " Zing"  }STUFF.TEXT: MYCG
```

GET.TEXT: (part# -- $string , return text for part)

PUT.TEXT: ($string part# -- , specify text for a part)

This will display the new text if called while the control is drawn. If you are defining the text for all of the parts, use }STUFF.TEXT: . Remember to define the text inside a colon definition so that it is permanent. (See **Debugging Controls** at the end of this chapter.)

OB.MENU.GRID subclass of OB.CHECK.GRID

The only difference between *Check Grids* and *Menu Grids* is that menu grids turn off as soon as you lift the mouse button. This makes them suitable for command buttons. All of the methods are the same.

OB.RADIO.GRID subclass of OB.CHECK.GRID

The only difference between Check Grids and Radio Grids is that Radio grids will turn off all other parts when you turn on one part. This makes them suitable for selecting "One of Many", like buttons on a car radio that select one station.

PUT.VALUE: (value part# -- , set value for part)

If this value is TRUE, the value for the other parts will be set to FALSE. This is a good way to set an initial value for a part of a Control.

OB.COUNTER subclass of OB.CONTROL

This class of control is useful for scrolling through a set of choices. The choices can be displayed as numbers or text strings. The *Shape Selector* in the Shape Editor is an example of an OB.COUNTER control.

When you click on the UP or DOWN arrows, the value of the control is incremented or decremented. The resulting value is passed to the DOWN function for processing. It is also passed to a special

function called the *Text Function*. This function returns text for a given value. It must have the following stack diagram:

```
TEXT.FUNC   ( value  -- addr count  )
```

The address returned must be the address of the first character of a text string. An example of a very simple text function is one that uses a CASE statement to select the text. In this example, we select the name of a synthesizer.

```
: TEXT.FUNC  ( value -- addr count )
   CASE
      0 OF " Moog" ENDOF
      1 OF " Serge" ENDOF
      2 OF " PAIA" ENDOF
      " Undefined!" SWAP  ( in case VALUE is > 2 )
   ENDCASE
   COUNT  ( convert $STRING to ADDR COUNT )
;
2 TEXT.FUNC TYPE  ( prints SERGE )
```

Another way to implement a predefined text function like the one above is to use the TEXTROM tool. Try this:

```
TEXTROM  SYNTH.NAMES ," Moog"  ," Serge" ," PAIA"
2 SYNTH.NAMES TYPE
```

A more common form is to use the VALUE of a part to index into a list of functions or objects, and return its name. Here is the definition of the Text Function used in the Shape Editor's Shape Selector control.

```
: SE.TEXT.FUNC  ( index -- addr count )
   GET: SHAPE-HOLDER  ( get shape from holder )
   GET.NAME: []  ( -- $name )
   COUNT  ( -- addr count )
;
```

The default *Text Function* is N>TEXT which converts a number, the value in this case, to a text string. The *Dimension Selector* of the Shape Editor uses this default.

Note: OB.COUNTERs do *not* support the **PUT.TEXT:** or **}STUFF.TEXT:** methods.

The parts of the Counter control are:

```
0 = top arrow
1 = top of text box
2 = bottom of text box
3 = bottom arrow
```

You can specify how many choices are possible by setting the MIN and MAX. If you want N choices, set the MAX to N-1. The choices will then be 0,1,2...N-1. For example, for the three synthesizers, we would set MAX to 2 so we could get 0,1 or 2. When you increment past the limit, the value automatically "wraps around" in either direction. The default values are 0 and 127.

GET.TEXT.FUNCTION: (-- cfa)

PUT.TEXT.FUNCTION: (cfa -- , specify text function)

OB.NUMERIC.GRID subclass of OB.CONTROL.GRID

Numeric Grids provide a way to alter a table of numbers. Each part can have its own VALUE, MIN and MAX. These are very handy for MIDI patch editors. You can change a value by clicking on the

top or bottom of the number to increment or decrement it. You can also click on a number and drag the mouse to continuously change the value.

The grid will be labelled along the *left side* using either the *Text Function* specified with **PUT.TEXT.FUNCTION:** or the text specified using **PUT.TEXT:** or **}STUFF.TEXT:**. The Text Function has a higher priority. The index of the text will be 0, 1, 2 .. NUMY-1. Thus for a 2x5 grid there would be 5 labels numbered 0,1,2,3,4.

OB.FADER subclass of OB.CONTROL

Faders are like the slide pots on an audio mixer. You can grab a knob and move it up or down to change the value. If you click above or below the knob it will add or subtract the specified increment value. They are nice because they give a graphical indication of their value. You can look at a row of faders, and immediately see what their values are by the positions of the knobs.

The resolution of faders is limited to the pixel resolution of the display. With a MIDI velocity Fader that is 40 pixels high, then you will be able to specify only 40 different velocities. If you need high resolution, use a numeric controller, or use a tall fader.

If the fader code is not already loaded, use:

```
INCLUDE? OB.FADER  H:CTRL_FADER
```

The NEW: method does not need to be called for faders.

GET.KNOB.SIZE: (-- height)

PUT.KNOB.SIZE: (height -- , set height in WC)

The height of the knob is specified in world coordinates, 0-4095. If the size is zero, the knob will not be drawn.

IF.SHOW.VALUE: (flag -- , show value when changed?)

Faders, by default, will continuously display their value below the strip. Call this method with a FALSE to turn this off.

PUT.INCREMENT: (n -- , amount to add or subtract)

Used when clicked above or below knob.

OB.XY.CONTROLLER subclass of OB.NUMERIC.GRID

An XY Controller has a knob that you can move in two dimensions, horizontally and vertically. Each dimension is considered a *part*.

```
Horizontal    = part #0
Vertical      = part #1
```

Important: XY controllers always have 2 parts and they usually change together. The DOWN, MOVE, and UP functions, therefore, have a different stack diagram then the other controls. Instead of being passed the VALUE and a PART#, they are passed:

```
XYFUNC  ( x-value y-value -- )
```

You can disable either part to allow motion in one direction using PUT.ENABLE: .

If this class is not already loaded, load it by entering:

```
INCLUDE?  OB.XY.CONTROLLER  H:CTRL_XY
```

NEW: (-- , allocates memory for 2 dimensions)

GET.KNOB.SIZE: (part# -- size)

PUT.KNOB.SIZE: (size part# -- , set size)

[Notice that this is different than the OB.FADER methods.] Set the size for a given dimension/part.

```
If SIZE > 0 then SIZE in World Coordinates
If SIZE = 0 then don't display knob.
If SIZE < 0 then SIZE in Device Coordinates
```

VALUE>X: (value -- x , pixel position of center of knob)

This method is handy for determining the x position corresponding to a given value in the horizontal dimension. This could be used to generate an overlay to guide the user in positioning the knob.

VALUE>Y: (value -- y, pixel position of center of knob)

X>VALUE: (x -- value , value for that position)

XY.ONLY.HORIZONTAL (control -- , disable vertical)

This will take an xy control and make it a purely horizontal fader. It does this by disabling the vertical part and setting the vertical knob to the full height. This must be called **after** NEW: and PUT.WH: .

XY.ONLY.VERTICAL (control -- , disable horizontal)

Similar to XY.ONLY.HORIZONTAL.

XY_HORIZONTAL_PART (-- 0 , constant)

XY_VERTICAL_PART (-- 1 , constant)

Y>VALUE: (y -- value , value for that position)

OB.TEXT.GRID subclass of OB.CONTROL.GRID

This control provides a grid of tiny text editors. These can be used to enter names or to enter numbers precisely. You can filter the characters input to the grid using a custom filter function. You can specify a function to be called when you hit a carriage return, or when you leave a particular part and begin to edit another.

IMPORTANT! Use NEW: to specify maximum text length. The text will also be limited to the size allowed by each parts box.

GET.CR.FUNCTION: (-- cfa)

GET.FILTER.FUNCTION: (-- cfa)

GET.JUSTIFY: (-- justification)

GET.LEAVE.FUNCTION: (-- cfa)

GET.TEXT: (part -- $text)

GET.VALUE: (part -- value)

This will attempt to convert the text in a part to a single precision number and return that number. You are probably better off using GET.TEXT: and NUMBER? so that you can handle bad numbers easier.

KEY: (char --)

This is called internally by the screen when a key is hit on the keyboard.

NEW: (numx numy maxchars --)

Allocate memory for parts for internal text storage. Note this takes 3 parameters while most controls take 2.

PUT.CR.FUNCTION: (cfa | 0 --)

Calls this function when the user hits a carriage return. The function must have the following stack diagram.

```
your.cr.function ( $text part -- )
```

Here is an example of a function that converts the text to a double precision number and prints it.

```
: SHOW.VALUE ( $text part -- )
```

```
      ." Part = " . cr
      ." Value = " number?
      IF
         d.
      ELSE
         ." Bad!"
      THEN
      cr
;
```

To convert a double precision number to single precision just DROP the high order cell.

PUT.FILTER.FUNCTION: (cfa | 0 --)

As each character is hit, it will be passed to this function. Your function must have the following stack diagram.

```
your.filter.function ( char -- ok? , true if ok )
```

Here is an example of a function that only allows hexadecimal number entry.

```
: PASS.HEX  ( char -- ok? , filter characters for number )
   >r
   r@ isdigit  \ 0-9?
   r@ toupper ascii A ascii F within? OR \ A-F ?
   r@ isprint not OR \ pass control characters like BACKSPACE
   rdrop
;
```

PUT.JUSTIFY: (0|1|2 --)

Text within each part will be justified as follows:

```
0 = left  \ nice for normal text
1 = center
2 = right  \ nice for numbers
```

PUT.LEAVE.FUNCTION: (cfa | 0 --)

This is called when you finish editing a part and click in any other control or another part of the same control. The function has the same requirements as the CR function.

PUT.TEXT: ($text part --)

Copies text into part using $MOVE. Since this makes a copy you don't have to preserve the text yourself. It can be on the PAD.

PUT.VALUE: (value part --)

This will convert N to text and call PUT.TEXT:

OB.SCREEN subclass of OB.ELMNTS

Screens contain a group of Controls. When a Screen is drawn, it draws its Controls. An example is the Shape Editor Screen.

Screens have 3 dimensions. Dimension 0 contains the *address* of the Control object. Dimensions 1 and 2 contain the *x,y coordinates* of the top left corner of that Control. Since Screens specify the placement of their Controls, a control can be shared between two different screens that place it in two different locations. If you don't want the screen to specify the location of the Control, set X and Y to (-1). To see an example of what's inside a screen, PRINT: the Shape Editor Screen. Enter:

```
PRINT: SE-SCREEN
```

You can specify a shortcut-key command using the **PUT.KEY:** method. This will allow you to quickly get to a screen without having to pull down the Custom menu. Just the special menu shortcut key combination for your computer.

You can put a screen inside another screen to build **hierarchical screens**. It will be treated like a *control* inside the higher level screen.

ADD: (control x y -- , add control to screen)

The top, left corner of the control will be at x,y. If x and y are -1, the control will be drawn at the location specified using the control's PUT.XY: method.

?DRAWN: (-- flag , true if currently drawn)

DUMP.SOURCE: (-- , print source code)

This method is used by the Screen Editor to print the source code necessary to create the Screen.

DRAW: (-- , draw all held controls)

This will also execute the DRAW.FUNCTION.

FREEALL: (-- , free: all held controls)

GET.DRAW.FUNCTION: (-- cfa , function to call when drawn)

GET.KEY: (-- key , command key ASCII character)

GET.TITLE: (-- $string)

GET.UNDRAW.FUNCTION: (-- cfa , function to call when undrawn)

NEW: (N 3 -- , allocate space for controls)

When a screend is NEW:ed, it is automatically added to a list called CUSTOM-SCREENS. When the HMSL window is opened, these screens are placed in the Screens Menu. See ADD:.

PUT.DRAW.FUNCTION: (cfa -- , function to call when drawn)

PUT.KEY: (key -- , command key ASCII character)

This will be used as a command key shortcut as an alternative to selecting the screen from the menu.

PUT.TITLE: ($string -- , set title for screen and menu entry)

PUT.UNDRAW.FUNCTION: (cfa -- , function to call when undrawn)

When a new screen is drawn, the old screen is "UNdrawn".

DEFAULT-SCREEN (-- addr , VARIABLE , not a method)

This variable contains the address of the Screen to draw when HMSL starts up. The Shape Editor will normally be drawn first but you can select your screen by setting this variable.

```
MY-SCREEN  DEFAULT-SCREEN  !
```

Special Topics

Screen Editor

Positioning controls on a Screen so they do not overlap can be a tedious process. Changing the numbers in a file, then compiling it and drawing the Screen over and over can take a while. Planning the layout ahead of time on a piece of graph paper can save time but there are always little adjustments that need to be made. For this purpose we have provided an interactive Screen layout program called the **Screen Editor**. Here are the steps to using the Screen Editor.

1) Compile Screen Editor by entering:

```
INCLUDE?  EDIT.SCREEN  HT:SCREEN_EDITOR
```

2) Design, code, and compile your screen and controls using rough x,y positions.

3) Edit the Screen by passing the address of your Screen object to EDIT.SCREEN A window will open and your Screen will be drawn. As an example, let's use the screen from the tutorials above, enter:

```
MYSC.INIT   ( initialize Screen and Controls )
MY-SCREEN  EDIT.SCREEN
```

4) You can now change the x,y position of the Controls as well as their width and height. In the Screen Editor, each Control is divided into four quadrants. By clicking and dragging on a quadrant you can change a parameter. Here is a description of each quadrant and what it affects.

```
Top-Left        => x,y position
Top-Right       => width
Bottom-Left     => height
Bottom-Right    => width and height
```

5) When you are done editing, click on the window's close box. The Screen Editor will call: DUMP.SOURCE: for the screen which will print the source code needed to recreate the screen layout. (On the Macintosh, you can type "DUMP.SOURCE: screenname" into the HMSL editor then hit <ENTER>. This will cause the output to be put directly into your file.) If you want you can use LOGTO to send this output to a file. You can then copy the code directly into your program.

```
LOGTO  filename   ( file to be created )
DUMP.SOURCE:  MY-SCREEN
LOGEND
```

Now edit the file you created, cut the code from that file and put it in your program.

Using Precompiled Screens

If you develop some screens that you use frequently, you can add them permanently to your HMSL4th image. Follow these steps:

1) Write **USER.INIT** and **USER.TERM** words to initialize your screens.

HMSL.INIT will search the dictionary for the top occurrence of **USER.INIT** and call it. You can initialize several screens, (or other systems), by chaining these calls together. Using the example from the begining tutorial, you would add these calls to the end of your screens file.:

```
: USER.INIT  ( -- )  USER.INIT  MYSC.INIT ;
: USER.TERM  ( -- )  MYSC.TERM  USER.TERM ;
```

2) Run HMSL but do NOT initialize it.

2) Compile your screen(s).

4) **SAVE-FORTH** using the instructions in your machine specific supplements.

The Shape Editor and Action Screen are made part of HMSL using this technique.

Debugging Controls

- **Text in Controls is garbled or changes** — The text used in controls must be defined inside colon definitions for them to be permanent. Text defined outside a colon definition is placed on the PAD, a temporary holding area which changes frequently.

```
" Pitch"  PUT.TITLE:  PITCH-CONTROL   ( This WON'T work )
: BUILD.PITCH.CG  ( -- , initiatlize control )
   " Pitch"  PUT.TITLE: PITCH-CONTROL ( This WILL work)
   ( other setup code )
;
```

- **"Stack Depth Change" error when you click on a Control.** — Your DOWN, MOVE or UP function is leaving something on the stack or eating too much. Test each of your functions by calling them with an appropriate VALUE and PART#. Make sure that they don't leave anything on the stack

or eat too much from the stack. If your functions have conditionals, i.e. IF...THEN or CASE, test each possibility. Missing an "ELSE DROP" is very common. The stack diagram should be:

```
MYFUNC  ( value part# -- )
```

- **Screens disappear from Screens menu.** — When a screen is NEW:ed its address is placed in the CUSTOM-SCREENS list. This list determines what screens show up in the Screens Menu. If the CUSTOM-SCREENS list accidentally gets cleared, you can add screen back using the ADD: method. Here is an example of adding the Shape Editor screen back in.

```
SE-SCREEN ADD: CUSTOM-SCREENS
```