expert sleepers disting NT

Lua scripting

Version 1.11

We are programmed just to do
Anything you want us to

- *Kraftwerk, “The Robots”*

Table of Contents

Introduction

The scripting language Lua[1] is embedded into the disting NT. It can currently be accessed in the following ways:

- Via the Lua Script algorithm.
- Via the UI scripts.
- Via the Lua console tool.

These are explored in more detail below.

In all cases, the scripts or user input are interacting with a single ‘instance’ of Lua on the machine. So, for example, global variables are accessible across all scripts. Similarly, however you use Lua on the disting NT you have access to the same additional functions that the module defines.

Lua version

The version of Lua implemented in the disting NT is currently 5.4.6.

Libraries

The use of the `require` keyword to load libraries is supported. The search path is

```
/programs/lua/?;/programs/lua/?.lua;/programs/lua/lib/?;/programs/lua/lib/?.lua
```

The recommend location for libraries is `/programs/lua/lib/`

For example

```
require 'complex'
local c = complex.add( complex.i, complex.new( 10, 20 ) )
```

1 https://www.lua.org

The Lua Script algorithm

Introduction

The Lua Script algorithm allows you to run what are effectively scripted “plug-ins”, loaded from the MicroSD card. These scripts run alongside the module’s built-in algorithms, and are part of the core audio/CV bus processing system.

Please refer to the disting NT user manual for information on installing scripts and loading the algorithm.

Anatomy of a script

A simple script might look like this:

```
-- LFO
-- Simple LFO example.

local t = 0.0

return
{
	name = 'LFO'
,	author = 'Expert Sleepers Ltd'

,	init = function( self )
		return
		{
			inputs = 1
		,	outputs = 2
		}
	end

,	step = function( self, dt, inputs )
		local f = 1 + inputs[1]
		t = t + dt * f
		if t >= 1.0 then
			t = t - 1.0
		elseif t < 0.0 then
			t = t + 1.0
		end
		local sqr = t > 0.5 and 5.0 or -5.0
		local tri = 20 * math.min( t, 1 - t ) - 5
		return { sqr, tri }
	end

}
```

The return value from the script is a table, most of the elements of which are functions. The system will call these functions at various times to initialise and execute the algorithm.

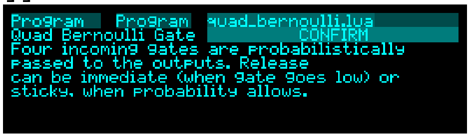
Name and description

The first two lines of the example above are comments, which are picked up by the module and used to present information to the user before the script is actually loaded.

The first comment is the script name, which should be fairly short.

The second comment is the description, which can be quite long. You can optionally use the Lua multi-line comment syntax `--[[ ]]`. For example:

```
-- Quad Bernoulli Gate
--[[
Four incoming gates are probabilistically passed to the outputs. Release
can be immediate (when gate goes low) or sticky, when probability allows.
]]
```



The script is a chunk

The script is loaded and executed by the Lua system on the module. As such, it is a chunk[2], and can have local variables, local functions etc.

In the example above, the variable t is local to the script.

The 'self' table

The functions that implement the script all have `self` as their first parameter. This holds the table that was returned when the script was initially executed. So for example in the script above, `self.author` will have the value 'Expert Sleepers Ltd'.

An alternative to using script-local variables is to add elements to this table. For example, rather than using the `local t`, the above script might choose to use `self.t`.

The 'init' function

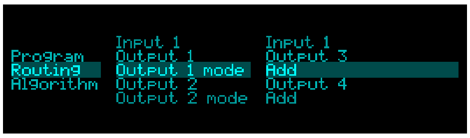
The `init` function is called once when the script is first loaded. Its main purpose is to return a table describing its inputs, outputs, and parameters. From the example:

```
return
{
      inputs = 1
,     outputs = 2
}
```

`inputs` and `outputs` can be anywhere from 0 to 28 – the number of busses on the system. If either is omitted, it is taken as 0.

The algorithm exposes parameters automatically to allow routing of the script's inputs and outputs to the system busses:

2 https://www.lua.org/manual/5.3/manual.html#3.3.2



The 'step' function

The `step` function is called regularly – in the current firmware, every 1ms. It is the script's opportunity to read from the input busses and write to the output busses.

```
,     step = function( self, dt, inputs )
            local f = 1 + inputs[1]
...
            return { sqr, tri }
      end
```

`dt` holds the time step (in seconds) since `step` was last called.

`inputs` is an array (1-based) containing the bus voltages. It will have the number of elements as requested in `init`.

The return value from the function should be a table containing the output voltages. Note that it does not have to contain every element; only the ones that need to be updated. For example, you could use

```
            local outs = {}
            outs[2] = tri
            return outs
```

which would only update the second output; the first output would retain its previous value.

Stepped and linear outputs

By default, all outputs from scripted plug-ins are stepped, only updating once per `step` call. You can choose to have them linearly interpolated, if a smooth CV is more appropriate. You do this in the `init` call, for example:

```
            return
            {
                  inputs = 1
            ,     outputs = { kStepped, kLinear }
            }
```

So rather than `outputs` being an integer, it is now an array, the size of which determines the number of outputs. Each entry in the array can be one of `kStepped` or `kLinear` to determine how that output should be handled. For our LFO example, `kStepped` is appropriate for the square output and `kLinear` is better for the triangle output.

Triggers and gates

To avoid having to perform these common operations in Lua, inputs can be monitored for triggers and gates by the system, only calling back to the Lua script when something changes. This is much more efficient than doing the same thing in `step`.

Consider this example script (SRflipflop.lua in our GitHub):

```
return
{
	name = 'SRflipflop'
,	author = 'Expert Sleepers Ltd'

,	init = function( self )
		return
		{
			inputs = { kTrigger, kTrigger }
		,	outputs = 1
		}
	end

,	trigger = function( self, input )
		self.state = input > 1
		local v = self.state and 5.0 or 0.0
		return { v }
	end

,	draw = function( self )
		drawText( 100, 40, self.state and "High" or "Low" )
	end

}
```

You will see that in the return from `init`, `inputs` is now an array rather than an integer. The size of the array determines the number of inputs, and the values in the array determine their type – one of `kCV`, `kGate`, or `kTrigger`.

In our example we have two trigger inputs. When the trigger fires, the system calls the `trigger` function in the script. The `input` parameter to this function tells the script which input caused the trigger.

The return from `trigger` is a table of output voltages, exactly as from `step`. Again, this table can be sparse; it doesn't have to contain every output voltage that the script defines.

Similarly, adsr.lua uses a gate:

```
return {
    inputs = {kGate},
    outputs = {kLinear},
}
```

Analogous to the `trigger` function, gates use a `gate` function:

```
gate = function(self, input, rising)
```

Again, `input` is an integer specifying which input gate changed, and `rising` is a boolean value specifying whether the gate has just opened (true) or closed (false).

The `gate` function should return a table of outputs to update, exactly as `step` and `trigger`.

Input and output naming

The table returned from `init` can optionally include custom names for the inputs and outputs, which will otherwise be "Input 1", "Input 2", etc. For example:

```
        inputs = { kCV, kTrigger, kGate }
,       inputNames = { [2]="Trigger input" }
,       outputs = 2
,       outputNames = { "X output", "Y output" }
```

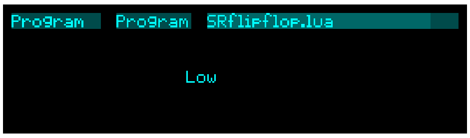
`inputNames` and `outputNames` are tables, indexed by the input and output numbers. In the example above, no custom name is given for inputs 1 & 3, so these will use the default “Input 1” and “Input 3”.

The ‘draw’ function

The previous (SRFlipFlop) example also introduces `draw`. This function is called regularly (30fps in the current firmware) and allows the script to define its own custom display.

Any of the `drawXXX` functions described below can be used within `draw`.

If the function returns nothing (or anything that Lua considers equivalent to boolean false), the standard parameter line will be drawn at the top of the screen. For example, the above SRFlipFlop `draw` produces this:



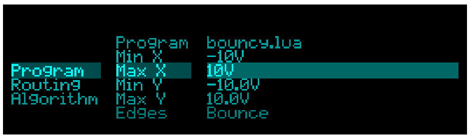
If `draw` returns boolean true, the top line is suppressed.

Parameters

Scripts may also define algorithm parameters, in the return from `init`. For example, from ‘bouncy.lua’:

```
return
{
        inputs = { kCV, kTrigger, kGate }
,       outputs = 2
,       parameters =
        {
                { "Min X", -10, 10, -10, kVolts }
        ,       { "Max X", -10, 10,  10, kVolts }
        ,       { "Min Y", -100, 100, -100, kVolts, kBy10 }
        ,       { "Max Y", -100, 100,  100, kVolts, kBy10 }
        ,       { "Edges", { "Bounce", "Warp" }, 1 }
        }
}
```

which results in this:



`parameters` is an array, each element of which is also an array. Three variants of the parameter array are shown above. The first is

```
,     { "Max X", -10, 10,  10, kVolts }
```

In this case the fields are: name, minimum value, maximum value, default value, unit. The values are integers (and will be passed back to the script as integers). The ‘unit’ takes one of these values:

```
kNone
kDb
kDb_minInf
kPercent
kHz
kSemitones
kCents
kMs
kSeconds
kFrames
kMIDINote
kMillivolts
kVolts
kBPM
```

The second parameter variant takes a scale value:

```
,     { "Max Y", -100, 100,  100, kVolts, kBy10 }
```

The scale can be one of `kBy10`, `kBy100`, or `kBy1000`. The minimum, maximum, and default values are divided by the scale, and handled as floats. For example the default above is 10.0V.

The third form a parameter can take is an enum value:

```
,     { "Edges", { "Bounce", "Warp" }, 1 }
```

The first field is the name, as always. The second field is an array of enum values. The third is the default value – 1 in this case referring to the first enum value, ‘Bounce’.

Within the script the parameter values can be accessed as `self.parameters`. For example in the above script, the ‘Edges’ parameter is `self.parameters[5]`. Note that this is read only access – changing parameter values should be done via `setParameter` (below).

self.algorithmIndex

The system adds a member to the script table called `algorithmIndex`, which is the index of the algorithm in the preset. It can be used as the argument to calls to, for example, `getCurrentParameter`.

self.parameterOffset

The parameters that a script defines are in addition to the parameters that the system maintains for the algorithm – the program choice, the routing etc.

All of the global functions that relate to parameters (e.g. `setParameter`) use an indexing that includes the system parameters, whereas the `self.parameters` array includes only the script-defined parameters.

The system adds a member to the script table called `parameterOffset`, which relates the two numbering systems.

An example of this is in the ‘bouncy.lua’ script:

```
,	draw = function( self )
		local alg = self.algorithmIndex
		local p = getCurrentParameter( alg ) - self.parameterOffset
		drawRectangle( cx, ty, cx, by, 1 )
		drawRectangle( lx, cy, rx, cy, 1 )
		local x1 = toScreenX( self.parameters[1] )
		local x2 = toScreenX( self.parameters[2] )
		local y1 = toScreenY( self.parameters[4] )
		local y2 = toScreenY( self.parameters[3] )
		drawRectangle( x1, y1, x2, y1, p == 4 and 15 or 2 )
```

The local variable `p` is the current parameter index in the script-relative numbering.

Custom UI

As well as presenting a custom display via `draw`, it is also possible to override the standard UI (the standard UI being the three pots to control parameter page, parameter selection, and parameter value).

To do this, give the script a function called `ui` which returns true:

```
,	ui = function( self )
		return true
	end
```

An example can be seen in the ‘bouncy.lua’ script.

If the system is gets a true value from `ui`, it does not implement the standard UI, and instead calls functions which may be (optionally) defined by the script. These functions are:

```
pot1Turn
pot2Turn
pot3Turn
encoder1Turn
encoder2Turn
pot3Push
pot3Release
encoder2Push
encoder2Release
```

These are identical to the functions that UI scripts can call, and are documented below.

A script may also define a `setupUi` function, which will typically be required only if the script defines a custom behaviour for the pots. This function is called whenever the algorithm’s UI appears for the first time (for example, when you switch from the overview display to the algorithm display).

`setupUi` takes one argument (`self`) and should return an array of pot positions (in the range 0.0-1.0). For example:

```
  pot1Turn = function( self, x )
    self.foo = math.floor( x * 100 + 0.5 )
   end,
  setupUi = function( self )
    local table = {}
    table[1] = self.foo/100.0
```

```
    return table
  end,
```

This allows the system to sync up the pot positions so that soft value takeover works.

Serialisation

Scripts may store arbitrary data in the module's preset files (as well as their parameter values, which are all handled automatically).

To do so, give the script a function called `serialise`, and return a table of data to be stored. The preset files are JSON, so the data needs to be JSON-friendly – numbers, strings, and booleans, arranged in tables or arrays. For example:

```
,     serialise = function( self )
            local state = {}
            state.testInt = 42
            state.testNum = 0.5
            state.testBool = true
            state.testArray = { 4, 8, 16 }
            state.testArray2 = { "ham", "eggs" }
            state.testObject = { red=1, green=7 }
            state.testComplex = { arr={10,9,8}, obj={a="low",b="high"} }
            return state
      end
```

will produce the following in the JSON:

```
   "state": {
   "testBool": true,
    "testNum": 5.000000e-01,
    "testComplex": {
    "obj": {
     "a": "low",
      "b": "high"}
     ,
     "arr": [10,9,8]
     }
    ,
    "testInt": 42,
    "testArray": [4,8,16]
    ,
    "testArray2": ["ham","eggs"]
    ,
    "testObject": {
    "green": 7,
     "red": 1}
    }
```

When the preset is loaded, the state table is loaded and stored in the script's `self` table, just before `init` is called. So, within `init` the loaded state is available as `self.state`. You're free to process this into other data structures, or simply leave it where it is and use it later.

MIDI

Scripts may use the `sendMIDI` function to send MIDI messages.

They may also define a function to be called to receive MIDI. Since this could potentially cause a large processing overhead, the interface to this is designed to allow messages to be filtered in the native C code before calling into Lua.

To receive MIDI, add a `midi` member to the table returned from `init`:

```
,     parameters =
      {
      ,     { "Edges", { "Bounce", "Warp" }, 1 }
      ,     { "MIDI channel", 0, 16, 0 }
      }
,     midi = { channelParameter = 2, messages = { "note", "cc", "bend", 
"aftertouch", "poly pressure", "program change" } }
```

The `midi` table has two members.

- `channelParameter` is the index in the parameters table of a parameter that allows the user to choose the MIDI channel on which the script should listen. (‘0’ turns off MIDI altogether.)

- `messages` is an array of MIDI message types that the script want to receive.

The system will then call a member function named `midiMessage` when a matching message is received:

```
midiMessage = function( self, message )
      if message[1] == 0x90 then
            lastMessage = "None on " .. message[2] .. " vel " .. message[3]
      elseif message[1] == 0x80 then
            lastMessage = "None off " .. message[2] .. " vel " .. message[3]
      end
end
```

,

UI scripts

Introduction

Please refer to the disting NT user manual for information on installing and running UI scripts.

Anatomy of a UI script

There are three main sections that a script needs to implement:

- Initialisation. The module calls this section once when the script is loaded to allow it to perform any required setup. This will typically include initialising any local state, and identifying the algorithms and parameters that the script will control.
- Responding to UI events. The script can choose to respond to any or all button pushes, encoders turns, pot turns etc.
- Drawing the UI. This can be as simple as showing the value of the parameter being controlled, or completely freeform vector graphics and text.

With that in mind, here is a simple example:

```
local augustus
local p_multiplier

return
{
	name = 'Example UI script'
,	author = 'Expert Sleepers Ltd'
,	description = 'controls one parameter of Augustus Loop'

,	init = function()
		augustus = findAlgorithm( "Augustus Loop" )
		if augustus == nil then
			return "Could not find 'Augustus Loop'"
		end
		p_multiplier = findParameter( augustus, "Delay multiplier" )
		if p_multiplier == nil then
			return "Could not find 'Delay multiplier'"
		end
		return true
	end

,	pot3Turn = function( value )
		setParameterNormalized( augustus, p_multiplier, value )
	end

,	button2Push = function()
		exit()
	end

,	draw = function()
		drawStandardParameterLine()
		drawText( 30, 40, "Hello!" )
	end
}
```

The return value from the script is a table, most of the elements of which are functions to handle various events. You can define some script-local variables before returning the table – here `augustus` and `p_multiplier` are such variables.

The `name`, `author`, and `description` elements are optional but encouraged.

The `init` function is called once when the script is loaded. In this case, the script takes the opportunity to search for and cache the indices of the algorithm and parameter that it would like to control. This is purely in the interests of efficiency – it could search every time it wanted to change the parameter. Or indeed, the script could just hard code the indices, but that would make it very brittle and likely to break if any changes were made to the preset that the script is designed to work with. The 'init' function returns true if everything is OK; if it doesn't, the module will abandon the script and revert to normal operation.

This example script watches for two UI events – turning pot 3, and pressing button 2. The latter causes the script to exit and return to the normal module UI; the former sets a parameter on the algorithm that was previously identified.

The final function `draw` is where the script gets to actually display something. `drawStandardParameterLine` draws the most recently changed parameter across the top of the screen, as in the default algorithm view.

All drawing must be performed from within `draw`. Calling any of the `drawXXX` functions from elsewhere in the script will cause unexpected results.

Functions that a script can define

init

Called once when the script is loaded.

Takes no arguments.

Return Boolean true on success, else a string indicating the cause of failure.

pot1Turn/pot2Turn/pot3Turn

Called when the relevant pot is turned.

One argument: a number in the range [0.0,1.0].

Returns nothing.

encoder1Turn/encoder2Turn

Called when the relevant encoder is turned.

One argument: a number which is +1 for clockwise movement or -1 or anticlockwise movement.

Returns nothing.

Button push/release functions

All take no arguments and return nothing. The following are defined:

button1Push, button2Push, button3Push, button4Push

button1Release, button2Release, button3Release, button4Release

pot1Push, pot2Push, pot3Push

pot1Release, pot2Release, pot3Release

encoder1Push, encoder2Push

encoder1Release, encoder2Release

draw

Called (continuously) to allow the UI to draw on the display.

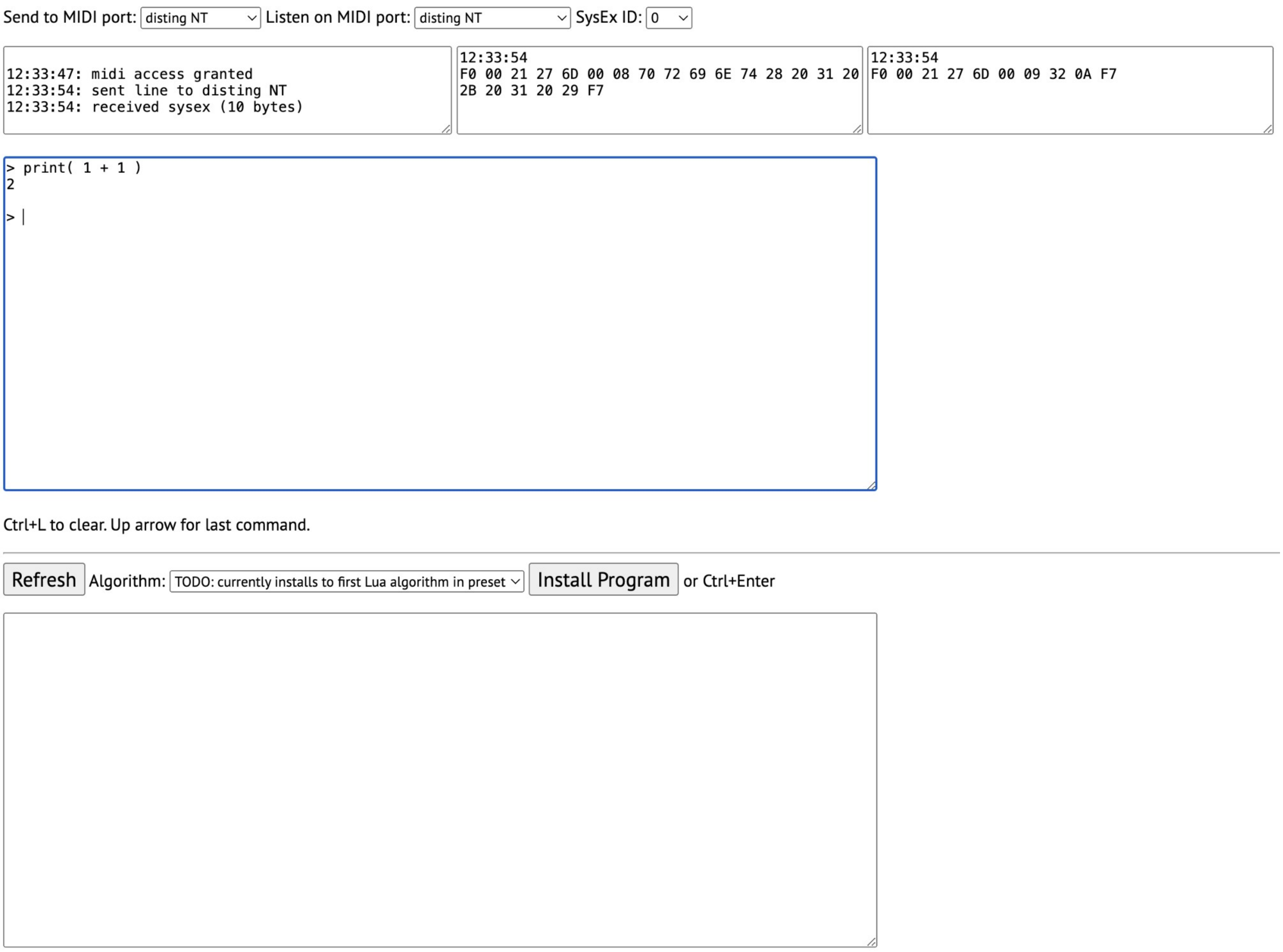
Takes no arguments; returns nothing.

The Lua console tool

The Lua console is a javascript program that runs in a web browser, communicating with the module via MIDI SysEx.

It can be found in our GitHub repository here[3].

It looks like this:



Interactive shell

The top section of the tool behaves somewhat like a standard bash etc. shell. You can enter commands, which are run immediately on the module, and the results returned.

You can use this to interact with the global Lua instance, shared by all scripts. For example:

```
> for k,v in pairs(_G) do if string.sub(k,1,4) == 'draw' then print( k, v ) end
end
drawSmoothBox     function: 0x600415ad
drawRectangle     function: 0x60041625
drawBox     function: 0x6004151d
drawText    function: 0x60094db5
```

3 https://github.com/expertsleepersltd/distingNT/tree/main/tools

```
drawSmoothLine    function: 0x600414a5
drawLine    function: 0x60041415
drawParameterLine function: 0x60094e11
drawStandardParameterLine     function: 0x60094e07
drawAlgorithmUI   function: 0x60041275
```

This queries the Lua globals and finds all the draw functions.

Possible uses for this include accessing global variables that affect debug functions in your scripts. It's also handy to simply check Lua syntax when coding.

Keyboard shortcuts

Enter submits the command for execution.

Ctrl+L clears the window.

The up arrow recalls the previous command so it can be issued again.

Shift+Enter adds a carriage return without executing the command, so you can type multi-line commands, for example:

```
> for i=1,4,1 do
print(i*i)
end
1
4
9
16
```

Installing programs

The lower section of the tool installs scripts into an instance of the Lua Script algorithm running on the module. It is provided so you can iterate on a program without having to constantly update and reload the version on the MicroSD card.

Note that it only updates the version loaded into the module's memory. Once your changes have been completed, you will need to copy the final version back to the card as usual.

At the time of writing the tool only updates the first Lua Script algorithm in the preset. It is anticipated that at a later time you will be able to choose which algorithm to update.

To begin, copy the entire Lua script into the box, and click 'Install Program' to install it. Ctrl+Enter is a keyboard shortcut to do the same thing.

Then make your edits, and install again to test.

The effect is as if you deleted the algorithm and reloaded it – the script starts with its `init` call etc. No state is preserved (unless you've written it into a global Lua variable, which is not encouraged).

Drawing in Lua scripts

Functions that perform drawing use pixels as their coordinate unit. The display is 256x64 pixels, with the origin (0,0) at the top left.

The display supports 16 shades; functions that take a colour argument use values from 0 (pixel off) to 15 (pixel fully lit).

Language extensions

In addition to the base Lua language features, the following functions are implemented on the disting NT.

Algorithm functions

- findAlgorithm
- getAlgorithmCount
- getAlgorithmName
- getCurrentAlgorithm

Parameter functions

- findParameter
- focusParameter
- getCurrentParameter
- getParameter
- getParameterCount
- getParameterName
- setParameter
- setParameterNormalized

UI functions

- standardPot1Turn/standardPot2Turn/standardPot3Turn

Drawing functions

- drawAlgorithmUI
- drawBox
- drawLine
- drawParameterLine
- drawRectangle
- drawSmoothLine
- drawStandardParameterLine
- drawText
- drawTinyText

Global functions

- exit
- getBusVoltage
- getCpuCycleCount
- sendI2CCommand
- sendI2CGetter
- sendMIDI
- setDisplayMode

Function documentation (alphabetical)

drawAlgorithmUI

```
drawAlgorithmUI( looper )
```

Draws the specified algorithm's custom GUI.

Takes one argument: the algorithm index.

Returns nothing.

drawBox

```
drawBox( 20, 40, 25, 45, 15 )
```

Draws a box (an unfilled rectangle).

Takes five arguments: top left x/y, bottom right x/y, and colour. Coordinates are converted to integer values before drawing.

Returns nothing.

drawCircle

```
drawCircle( 30, 10, 20, 15 )
```

Draws a circle.

Takes four arguments: centre x, centre y, radius, and colour. Coordinates are converted to integer values before drawing.

Returns nothing.

drawLine

```
drawLine( 30, 10, 100, 20, 15 )
```

Draws a line.

Takes five arguments: top left x/y, bottom right x/y, and colour. Coordinates are converted to integer values before drawing.

Returns nothing.

drawParameterLine

```
drawParameterLine( lfo, p_speeds[i], ( i - 1 ) * 10 )
```

Draws a line of information similar to that drawn by `drawStandardParameterLine`, but for a specific algorithm and parameter.

Takes three arguments: the algorithm index, the parameter index, and a y coordinate offset (from the default position at the top of the screen).

Returns nothing.

drawRectangle

```
drawRectangle( 21, 41, 24, 44, 1 )
```

Draws a filled rectangle.

Takes five arguments: top left x/y, bottom right x/y, and colour. Coordinates are converted to integer values before drawing.

Returns nothing.

drawSmoothCircle

```
drawSmoothCircle( 30, 10, 20, 15 )
```

Draws an antialiased circle.

Takes four arguments: centre x, centre y, radius, and colour, all of which can meaningfully be floating point values.

Returns nothing.

drawSmoothLine

```
drawSmoothLine( 100, 25.5, 30, 18.2, 8.3 )
```

Draws an antialiased line.

Takes five arguments: top left x/y, bottom right x/y, and colour, all of which can meaningfully be floating point values.

Returns nothing.

drawStandardParameterLine

```
drawStandardParameterLine()
```

Draws the standard algorithm parameter line at the top of the screen, as in the default algorithm view, showing the most recently modified parameter.

No arguments; returns nothing.

drawText

```
drawText( 30, 40, "Hello!", 8, "right" )
```

Draws a string on the display in the module’s standard font.

Takes three, four, or five arguments: the x and y coordinates, the string to draw, optionally a colour, and optionally an alignment. The y coordinate specifies the text baseline. The colour is a value from 0-15; if not supplied, 15 is used. The alignment is a string, one of "centre" or "right"; if not supplied, left alignment is used.

Returns nothing.

drawTinyText

```
drawTinyText( 30, 40, "Hello!", 15, "centre" )
```

Draws a string on the display in the module's tiny 3x5 pixel font.

Takes three, four, or five arguments: the x and y coordinates, the string to draw, optionally a colour, and optionally an alignment. The y coordinate specifies the text baseline. The colour is a value from 0-15; if not supplied, 15 is used. The alignment is a string, one of "centre" or "right"; if not supplied, left alignment is used.

Returns nothing.

exit

```
exit()
```

When called from a UI script, returns control to the normal module UI.

No arguments; returns nothing.

findAlgorithm

```
augustus = findAlgorithm( "Augustus Loop" )
```

Allows the script to look up an algorithm within the preset.

Takes one argument, a string. Currently this is matched against the algorithms' (customised) names.

Returns as many results as matches found. Each is a 1-based index into the list of algorithms.

findParameter

```
p_multiplier = findParameter( augustus, "Delay multiplier" )
```

Allows the script to look up a parameter within an algorithm.

Takes two arguments: a number, the algorithm index; and a string, which is matched against the parameter names.

Returns as many results as matches found. Each is a 1-based index into the list of parameters.

For algorithms with variable numbers of parameters (according to their specification), the string is matched against the base parameter name and the prefixed parameter name as seen in, for example, the preset editor tool. Say you have an LFO algorithm with two channels – the string 'Speed' will match against the parameter for all channels, so this function will return two results, but the string ‘1:Speed’ or ‘2:Speed’ will give you only the parameter for the specific channel.

focusParameter

```
focusParameter( augustus, p_multiplier )
```

Sets the current algorithm and parameter (the parameter that `drawStandardParameterLine`() will show).

Takes two arguments: the algorithm index, and the parameter index.

Returns nothing.

getAlgorithmCount

```
local count = getAlgorithmCount()
```

Returns the number of algorithms in the preset.

getAlgorithmName

```
local name = getAlgorithmName( alg )
```

Takes one argument: the algorithm index.

Returns the name of that algorithm (as displayed on the overview screen).

getBusVoltage

```
local v = getBusVoltage( 1, 12 )
```

Gets the voltage on a bus at an algorithm’s input. (These are the same voltages that are displayed in the algorithm overview display.)

Takes two arguments: the algorithm index, and the bus index (zero-based). The algorithm index value ranges from zero to the number of algorithms, the last value effectively returning the bus voltage at the last algorithm’s output.

Returns the voltage.

getCpuCycleCount

```
local count = getCpuCycleCount()
```

Returns the value of the CPU’s on-chip cycle counter. This can be used to estimate how long a section of code takes to run. Being a 32 bit counter at 600MHz, it overflows every 7 seconds, approximately.

getCurrentAlgorithm

```
local alg = getCurrentAlgorithm()
```

Returns the index of the current algorithm (that is, the one that would be highlighted in the algorithm overview screen).

getCurrentParameter

```
local p = getCurrentParameter( alg )
```

Takes one argument: the algorithm index.

Returns the index of the current parameter for that algorithm.

getParameter

```
local v = getParameter( augustus, p_multiplier )
```

Gets an algorithm parameter's value.

Takes two arguments: the algorithm index, and the parameter index.

Returns the value.

getParameterCount

```
local v = getParameterCount( alg )
```

Takes one argument: the algorithm index.

Returns the number of parameters that the algorithm has.

getParameterName

```
local name = getParameterName( alg, index )
```

Takes two arguments: the algorithm index, and a parameter index.

Returns the name of the indexed parameter within the algorithm.

sendI2CCommand

```
sendI2CCommand( 0x32, 0x46, 7, 0, 2 )
sendI2CCommand( 0x32, { 0x46, 7, 0, 2 } )
```

Sends an I2C command. The arguments are the I2C address to send to, followed by the command & data bytes. The command & data bytes may be included as simple parameters, or be contained in a table.

Returns nothing.

sendI2CGetter

```
local data = sendI2CGetter( 0x32, 2, 0x48, 7 )
local data = sendI2CGetter( 0x32, 2, { 0x48, 7 } )
```

Sends an I2C getter. The arguments are the I2C address to send to, the number of bytes expected in the response, followed by the command & data bytes. The command & data bytes may be included as simple parameters, or be contained in a table.

Returns an array of bytes.

sendMIDI

```
sendMIDI( where, 0x90, 48, 127 )
```

Sends a MIDI message.

The first argument is a bitmask of destinations, with the values

0x1 – MIDI breakout
0x2 – Select Bus
0x4 – USB
0x8 – Internal

Following this are one, two, or three arguments, which are the bytes of the MIDI message to send.

Returns nothing.

setDisplayMode

```
setDisplayMode( "overview" )
```

Sets the module’s display mode.

Takes one string argument, which specifies the new display mode. The options are "overview", "meters", "parameters", "ui" (the custom UI for the current algorithm), "algorithm" (the current algorithm’s parameters or UI depending on which was most recently used), and "menu".

Returns nothing.

setParameter

```
setParameter( augustus, p_multiplier, value, focus )
```

Sets an algorithm parameter’s value.

Takes four arguments: the algorithm index, the parameter index, the parameter value, and whether to focus the UI (the ‘current parameter’) on the changed parameter. `focus` is optional – if not provided it is assumed `true`.

Returns nothing.

setParameterNormalized

`setParameterNormalized( augustus, p_multiplier, value, focus )`

The same as `setParameter`, except the third argument is a number in the range [0.0,1.0], which is mapped onto the full range of the parameter being changed.

standardPot1Turn/standardPot2Turn/standardPot3Turn

`standardPot1Turn( value )`

Performs the standard function of the pot when in the algorithm view. Typically used as e.g.

```
,     pot1Turn = function( value )
            standardPot1Turn( value )
      end
```

Acknowledgments

Lua