cellsDS Developer's Guide

Like most Nintendo DS applications, the core of cellsDS is written in C++ and cannot be modified. Luckily all of the fun stuff is written in the Lua scripting language. With a little knowledge of Lua, you can modify the existing engines or create brand new engines. It's easy!

The Lua scripts are located at /cellsDS/scripts/

You can open and edit the .lua scripts using Notepad or whatever raw text editor you have handy.

A very simple example: pixel.lua

```
Line #1:    function stylus_newpress()
Line #2:          set_pan(X)
Line #3:          play_note(17-Y,16)
Line #4:    end
```

This 'engine' plays a sound when you touch the grid. The stylus 'X' position controls the panning and the stylus 'Y' position chooses the pitch.

Let's dissect pixel.lua…

Line #1: function stylus_newpress()

Here we are implementing stylus_newpress. The stylus_newpress function is automatically called by cellsDS whenever you touch the sequencer grid.

Line #2: set_pan(X)

Calling set_pan( ) will set the pan of the current engine's sound. set_pan( ) takes one numeric argument in the range of 1 (pan far left) to 16 (pan far right). In our example, 'X' represents the column of the grid that the stylus touched, which ranges from 1 to 16.

Where did the variable X come from? When you touch the sequencer grid, X is *injected* into your .lua script as a global variable before your stylus_newpress() function is called. Similarly, the global variable 'Y' is injected into your lua script and represents the row where the stylus touched the grid.

Other variables that are automatically injected into your .lua script by cellsDS are:

```
X
Y
PAD_HELD_UP
PAD_HELD_DOWN
PAD_HELD_LEFT
PAD_HELD_RIGHT
PAD_HELD_X
PAD_HELD_Y
PAD_HELD_A
PAD_HELD_B
STYLUS_HELD

SEQUENCER_STEP_16
SEQUENCER_STEP_32
BLOCK
FILENAME

WHITE
DARK_GRAY
LIGHT_GRAY
BLACK
DARK_BLUE
BLUE
LIGHT_BLUE
DARK_RED
RED
LIGHT_RED
DARK_GREEN
GREEN
LIGHT_GREEN
YELLOW
CYAN
MAGENTA

ACTIVE
ENGINE_ID
```

I'll discuss these later, but it's notable that the constants WHITE, DARK_GRAY, etc. are injected in to your Lua script to make things easy later on.

Line #3: play_note(17-Y,16)

Calling play_note( ) is the only way to play a sound using cellsDS. play_note is defined as: play_note(*note_number*, *volume*) where:

> *note_number* - Determines the frequency of playback. (note_number does a lookup in the file /cellsDS/scale.txt to figure out which frequency to play. By default, a note_number of 1 plays the lowest frequency. A note_number of 16 plays the highest frequency.)
>
> *volume* – Playback volume. Ranges from 1 (quiet) to 16 (loud).

As I mentioned above, the global Lua variable ‘Y’ is automatically injected into your Lua script before the stylus_newpress() function is called. It represents the Y-position (or row) of the cell touched by the stylus. It ranges from 1 (the top row) to 16 (the bottom row).

The top rows are supposed to play higher pitched notes, so a little math (17-Y) was necessary.

That covers our simple example.

Calling cellsDS functions

Let's delve a little deeper.

Besides play_note() and set_pan(), you can call the following cellsDS functions from within your Lua script:

set_cell(x,y,color) – set a cell's color
set_all_cells(color) – set all cells in the grid to a color (usually white)
set_column(column, color) – set a column of the grid to a color
set_row(column, color) – set a row of the grid to a color

clear_top_screen( ) – clears the top screen
display_text(text_string,x,y)
    – displays a string on the top screen at pixel position x,y

play_note(scale_table_index, volume) – play a note at a specific volume
play_frequency(frequency, volume) – play a note at a specific volume
set_pan(pan) – set the pan for the engine
load_sound(sound_number) – chooses the sound for playback

set_block(block_number) – sets the global 'BLOCK' variable
set_global_variable(variable_name, variable_value)
    – sets a global variable in all engines (see Appendix A)

midi_note(midi_table_index, volume, duration)
    – simple way to send a midi note out
send_midi_note_on(channel, note, volume)
    – send out a midi note-on message
send_midi_note_off(channel, note, volume)
    - send out a midi note-off message
send_midi_message(message, data1, data2)
    - send out a generic midi message

For a detailed description of each, refer to the Reference section at the bottom of this document.

Implementing callback functions

In the simple example presented at the beginning of this document, we implemented the function *stylus_newpress*. When any cell of the grid was touched, cellsDS called our *stylus_newpress* function which played a sound.

Our Lua script can react to other types of user input by implementing any of the following functions:

```
pad_newpress_left
pad_newpress_right
pad_newpress_up
pad_newpress_down
pad_newpress_a
pad_newpress_b
pad_newpress_x
pad_newpress_y

pad_released_left
pad_released_right
pad_released_up
pad_released_down
pad_released_a
pad_released_b
pad_released_x
pad_released_y

stylus_newpress
stylus_released
stylus_drag

on_load
clock
redraw
save_snapshot
load_snapshot
```

Here's another small Lua script to illustrate:

```
function pad_newpress_a()
      set_cell(X,Y,DARK_BLUE)
end

function pad_released_a()
      set_cell(X,Y,WHITE)
end
```

on_load(), clock(), redraw(), save_snapshot(), and load_snapshot()

Some very useful functions to implement in your Lua scripts are *on_load*, *clock*, and *redraw*.

on_load( ): is called by cellsDS after your Lua script is loaded but before any other functions are called. It's a great place to create and initialize global Lua variables.

The following is an example on_load function that's used by step_sequencer.lua. It creates and initializes a three-dimensional array which is used to store the sequencer's state:

```
function on_load()

      local block_count
      local i
      local j
      grid = {}

      for block_count=1,8 do
            grid[block_count] = {}
            for i=1,16 do
                  grid[block_count][i] = {}
                  for j=1,16 do
                        grid[block_count][i][j] = WHITE
                  end
            end
      end

      old_column = 1
      selected_sound = 1
      drag_color = WHITE

end
```

clock( ): is called by cellsDS when the sequencer advances 1 step. The sequencer timing is handled within cellsDS so you don't have to worry about it.

The global variable `SEQUENCER_STEP_16 is also helpful. It ranges from 1 to 16 and represents the current internal sequencer position in cellsDS. Here's an example using both the clock() function and the SEQUENCER_STEP_16 global variable:`

```
function clock()
      if SEQUENCER_STEP_16 == 1 then
            play_note(6,16)
      end
end
```

`The above example plays a note every time the sequencer advances to step 1.`

redraw( ): may be called by cellsDS at certain times, such as when your engine is selected as the active engine. If your engine has any type of visuals, it'd be a good idea to implement the redraw function.

Here's the redraw function from step_sequencer.lua:

```
function redraw()

      local i
      local j
```

```
        for i=1,16 do
              for j=1,16 do
                    set_cell(i,j,grid[BLOCK][i][j])
              end
        end
  end
```

save_snapshot( ) and load_snapshot( ): When someone saves their snapshot, cellsDS calls each Lua script's save_snapshot( ) function. If your Lua script is save-able, you should implement the save_snapshot( ) function to store any state information to a file. Ditto for load_snapshot( ).

The global variable FILENAME will be injected into your Lua script before save_snapshot or load_snapshot are called. It holds the filename which you should use when storing or retrieving your sequencer data.

For example, here's how step_sequencer.lua does it:

```
function save_snapshot()

        local i
        local j
        local block_number

        file = io.open(FILENAME,"w")

        -- write out grid
        for block_number=1,8 do
                for i=1,16 do
                        for j=1,16 do
                                if grid[block_number][i][j] ~= WHITE then

                                -- write out line that looks like: grid[15][10] = 4

                                        file:write( "grid[" .. block_number .. "]["
                                                .. i .. "][" .. j .. "] = " ..
                                                grid[block_number][i][j] .. "\n");
                                end
                        end
                end
        end

        -- write out instrument selection

        file:write("load_sound(" .. selected_sound .. ")\n");
        file:write("selected_sound = " .. selected_sound .. "\n");

        file:close()

end

function load_snapshot()

        local block_count
        local i
        local j

        for block_count=1,8 do
                for i=1,16 do
                        for j=1,16 do
                                grid[block_count][i][j] = WHITE
                        end
                end
        end

        dofile(FILENAME)
end
```

Performance considerations

I've spent many days trying to make cellsDS as fast as possible. However, it is essential to pay attention to performance when writting Lua scripts. A slow Lua script will cause the DS to intermittently ignore button presses and stylus presses.

What is slow? To give you an idea, let me share one of my experiences. I had a great idea for an engine: Blocks would fall from the top of the screen to the bottom in pairs. I called the engine "chord rain", because it was "raining chords". When the blocks hit the bottom of the screen, they'd trigger notes to be played. It would be akin to a matrix screen-saver. My clock() function contained the following:

```
for row=16,2,-1 do
        for column=1,16 do
                grid[column][row] = grid[column][row-1]
        end
end
```

The function above would effectively scroll the array toward the bottom. However, I immediately started seeing performance degradation. CellsDS started intermittently ignoring my stylus presses. I spent the new few days researching how to fix the issue but ultimately found that the DS just couldn't keep up. The moral of this story is: Your Lua script must be ultra fast. Avoid looping through large arrays. Avoid looping through two dimensional arrays.

I'm very unhappy that there's such a vicious CPU limitation. If I can solve it in the future, I certainly will. CellsDS would be a much better application if the Lua scripts had more time to do more interesting things. Simulating Cellular Automaton, for example, would be far too intensive and would cause performance issues.

Assorted Notes

- cellsDS is smart enough ***not*** to redraw cells that it doesn't have to. For example, if you call set_cell(1,1,WHITE), but the cell at 1,1 is already white, cellsDS won't bother redrawing the cell.

- Your Lua script does not have to implement clock, save_snapshot, on_load, or any other function.

- Lua functions having to deal with input handling, such as pad_newpress_left( ) and stylus_newpress( ) will only be called for the active engine.

- Calls to set_column( ), set_row( ), and set_cell( ) will only have an effect when called by the active engine. Calls to these functions will be ignored by cellsDS when issued by inactive engines. (In other words, only the active engine may draw on the grid.)

- The ACTIVE global variable is set to "1" for the active engine. You can use this variable in your scripts to avoid doing unnecessary calculations and improve performance.

  For example: `if ACTIVE then set_cell(SEQUENCER_STEP_16,i,DARK_BLUE) end`

Parting notes

I hope that this document has been helpful. I highly suggest that you check out the Lua scripts that are included with cellsDS.

Enjoy,
- Bret

Appendix A: Reference

color constants

```
0  BLACK
1  DARK_GRAY
2  MEDIUM_GRAY
3  LIGHT_GRAY
4  WHITE
6  DARK_BLUE
7  RED
8  GREEN
9  BLUE
10  YELLOW
11  CYAN
12  MAGENTA
```

other constants

`ACTIVE`

```
This Lua global variable is set to 1 when the engine has been
selected on the sequencer page, otherwise it is set to 0.
Checking this variable can be useful to avoid calling draw
function unnecessarily.

Example:

if ACTIVE then
      -- draw some stuff
end
```

`ENGINE_ID`

```
This Lua global variable is set to the slot where the engine is
loaded.  It is zero based.

Example:

if ENGINE_ID == 1 then
      -- do something only if this engine
      -- was loaded into the second slot
end
```

cellsDS functions

display_text (string text, int x, int y)

Displays a text string on the top screen at pixel position x,y.

Parameters:

text – The text to output.
x – The x position to display the text. Valid numbers are 0 through 255.
y – The x position to display the text. Valid numbers are 0 through 191.

Examples:

```
--- displays instructional message at the top left of the top screen
display_text("Change Sound: Hold [LEFT] and touch a cell",1,1)
```

clear_top_screen ( )

Clears the top screen to black.

Examples:

```
clear_top_screen( )
```

set_cell(int x, int y, int color)

Set a cell's color.

Parameters:

x – Grid column of the cell. Valid numbers are 1 through 16.
y – Grid row of the cell. Valid number are 1 through 16.
color – Color to set the cell

Examples:

```
--- sets the cell at position 4,5 to dark gray
set_cell(4,5,DARK_GRAY)

--- sets the cell at position 4,5 to medium gray
set_cell(4,5,2)
```

set_all_cells(int color)

Sets all cells in the grid to a single color (usually white).

Parameters:

color – Color to set the cell

Examples:

--- set all the cells on the grid to white
set_all_cells(WHITE)

--- also sets all the cells on the grid to white
set_all_cells(4)

set_column(int column, int color)

Set a column of the grid to a color.

Parameters:

column – The column containing the cells to set. From 1 to 16.
color – Color to set the cells in the column

Example:

--- set all the cells in the 3rd column to dark blue
set_column(3,DARK_BLUE)

--- set all the cells in the 12th column to 3 (LIGHT_GRAY)
set_column(12,3)

set_row(int row, int color)

Set a row of the grid to a color.

Parameters:

row – The row containing the cells to set. From 1 to 16.
color – Color to set the cells in the row

Example:

--- set all the cells in the 4th row to blue
set_row(4,DARK_BLUE)

--- set all the cells in the 6th row to WHITE (which is 4)

```
set_row(6,4)
```

set_pan(int pan)

Calling set_pan( ) will set the pan of the current engine's sound. set_pan( ) takes one numeric argument in the range of 1 (pan far left) to 16 (pan far right).

Parameters:

pan – The pan, ranging from 1 (far left) to 16 (far right)

Example:

```
--- center pan
set_pan(8)

--- set  pan to the far left
set_pan(1)

--- set pan based on the stylus X position
set_pan(X)
```

play_note(int scale_table_index, int volume)

Play a note at a specific volume using a look-up table to determine frequency.

Parameters:

note – The scale_table_index parameter determines the frequency to play the sound. scale_table_index is an index into a table of frequencies which is defined in /cellsDS/scale.txt. The note parameter ranges from 1 to 16, where 1 is the lowest frequency and 16 is the highest frequency. This was done to make it easy to switch the default global scale.

volume – Integer value ranging from 1 (quiet) to 16 (loud).

Examples:

```
--- play a note at middle C at full volume
play_note(8,16)

--- play a note based on the Y position of the stylus
play_note(17-Y,16)
```

play_frequency(int frequency, int volume)

Play a note at a specific volume based on an absolute frequency.

Parameters:

frequency – The frequency to play the note. Ranges from 0 to some really, really large number.

volume – Integer value ranging from 1 (quiet) to 16 (loud).

Examples:

```
--- play a note at 22000htz at full volume
play_frequency(22000,16)
```

load_sound(int sound_number)

Loads a sound for use in playback. Because of memory limitations in the DS, not all of the sounds are loaded into memory at once. You have to explicitly load a sound before playing it using play_note( ). At the moment, multi-samples are not supported. Calling load_sound is an time consuming process and care should be taken when calling load_sound while a song is being played to avoid slowing down the DS and causing lags in your timing.

Parameters:

sound_number – Ranges from 1 to 255. The sound number corresponds to one of the sounds located in /cellsDS/sounds. Sound are named:

sound1.raw
sound2.raw
etc..

Examples:

```
--- load sound45.raw
load_sound(45)

--- load a sound based on the X and Y position of the stylus
load_sound(X + ((Y-1)*16))
```

set_block(int block_number)

Sets the global 'BLOCK' variable. Blocks are used to switch between different states in a song. The BLOCK variable is shared between all Lua scripts. Any of the Lua scripts can change the BLOCK variable. The purpose behind this is to allow you, the programmer, to change between the song states in unusual ways. For example, you can create a Lua script that randomly changes the global BLOCK variable every 4 steps.

Parameters:

block_number – Ranges from 1 to some-very-large-number, but a suggested range is 1 to 8

Examples:

```
--- set the block to 3
set_block(3)

--- sets the block based on the stylus X position
set_block(X)
```

set_global_variable(string variable_name, int value)

Sets a global variable in all 6 of the engines to the value supplied. There are no restrictions on the variable name, so be careful when using reserved words such as X, Y, BLUE, SEQUENCER_STEP_16, etc. Use this function to communicate between engines.

Parameters:

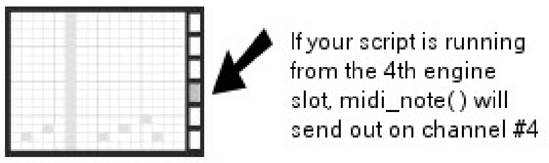
variable_name – a syntactically correct Lua variable name. For example, “trigger_count” or “fooBerry”, but probably not “$! locust ate ,my cat”

Examples:

```
--- sets the global variable “some_message” to 15 in all engines
set_global_variable(“some_message”,15)
```

midi_note(int midi_table_index, int velocity, int duration)

This is the easiest way to output a midi note. Instead of specifying the exact midi note value to output, midi_table_index is used, which retrieves the real midi note value from /cellsDS/midi.txt. The channel is derived from the engine slot (1 through 6) the script is loaded into.


If your script is running from the 4th engine slot, midi_note( ) will send out on channel #4

Parameters:

midi_table_index – midi_table_index is an index into a table of midi notes which is defined in /cellsDS/midi.txt. Ranges from 1 to 16.

velocity – Velocity is the note velocity which ranges from 1 to 16 (1=softest, 16=loudest).

duration – duration is the time, in sequencer steps, that the note will be held. A duration of '2' means that the note will be played, and after two sequencer steps a midi-note-off message will be sent for that note. (If the same note is played again before the duration is reached, the new note's duration will dictate when the midi-note-off message is sent.)

Example #1:

midi_note(5,16,2)

The preceding example immediately sends a midi-note-on message over WIFI.

The first parameter, midi_table_index, is an index into the /cellsDS/midi.txt file. Unless you've modified your midi.txt file, the resulting midi note number is '64'.

The second parameter, velocity, is set to 16. That's the maximum velocity. (Behind the scenes, the velocity is scaled to 0-127.)

Two sequencer steps after you issue this command, a midi-note-off message will be sent for this note on the correct channel.

Midi channel is determined for you. It's the slot that you've loaded in the Lua script making the call. The channel will range from 1 to 6.

This simple midi_note( ) function was created to make things easy on you. You don't need to stress about channels, note events, or other tricky stuff.

Example #2:

midi_note(17-Y,16,X)

The preceding example immediately sends a midi-note-on message over WIFI where the note value is controlled by the stylus Y position and the note duration is determined by the stylus X position.

send_midi_note_on(int channel, int note, int velocity)

Sends out a midi-note-on message over WIFI.

Parameters:

channel – midi channel number

note – Midi note number, ranging from 0 to 127

velocity – Midi note velocity, ranging from 0 to 127

Example:

send_midi_note_on(1,67,127)

The preceding example immediately sends a midi-note-on message over WIFI on channel #1. The note is 67, which is a 'G'

send_midi_note_off(int channel, int note)

Sends out a midi-note-off message over WIFI.

Parameters:

channel – midi channel number

note – Midi note number, ranging from 0 to 127

Example:

send_midi_note_off(1,67)

The preceding example immediately sends a midi-note-*off* message over WIFI on channel #1. The note to turn off is 67.

send_midi_message (int message, int data1, int data1)

Sends out generic midi message over WIFI.

Parameters:

message – midi message. Usually this will consist of a high and low byte OR’d together.
(See http://www.midi.org/about-midi/table1.shtml)

data1 – Midi data, ranging from 0 to 127

data2 – Midi data, ranging from 0 to 127