Documentation

Lumorphix

Table of Contents

1 Getting Started

The following section details the steps involved in getting started with the Lumorphix product, including the Lumorphix IP Core as well as evaluating a Lumorphix Beta release.

1.1 IP Core

The following page details getting started with the Lumorphix IP Core.

As Lumorphix is still in Beta, the Lumorphix IP Core has not yet been released!

1.1.1 Beta Release

See Beta Release (see page 6) to evaluate a Beta release of Lumorphix.

1.2 Beta Release

The following section details getting started with a Beta release evaluation - steps that are common across all beta releases. Begin with the Hardware (see page 7) setup section.

1.2.1 Hardware

The following page details the generic hardware setup involved in getting started with a Beta release of Lumorphix.

1.2.1.1 Setup

The generic hardware setup is pretty straightforward, and can be distilled to just two steps:

- If powered from a non-USB source, connect the hardware power supply.
- Connect the hardware to your PC via USB.

Any non-generic steps will be listed in the child page of Hardware Targets (see page 100) the corresponds to your specific hardware (see 'setup' within that page).

1.2.1.2 Next

See PC (see page 8) to setup your PC such that it can communicate with the target hardware.

1.2.2 PC

The following page details configuring your PC such that it can communicate with the target hardware.

1.2.2.1 Setup

To establish a connection between your PC and the target hardware follow the steps outlined below.

1.2.2.1.1 Linux or Mac OS

1. Check your FTDI kernel module is loaded (on Linux it is built into the kernel, so is likely already loaded):

```
user@pc> lsmod | grep ftdi_sio
```

3. If not, load it:

```
user@pc> sudo modprobe -v ftdi_sio
```

4. If that failed, see here[1] for installation instructions (VCP driver).
5. A USB device should now be present in **/dev/** :

```
user@pc>ls /dev/ |grep USB1
ttyUSB1
```

On some flavours of Linux, we've found that the device must be forced to baud 921600, after connecting the target hardware to your PC.

```
user@pc> stty -F /dev/ttyUSB1 921600
```

1 https://ftdichip.com/document/installation-guides/

1.2.2.1.2 Windows

1. If not already installed, see here[2] for installation instructions of the FTDI (VCP) driver.

1.2.2.2 Next

See Firmware (see page 10) to setup the Lumorphix firmware.

2 https://ftdichip.com/document/installation-guides/

1.2.3 Firmware

The following page details the generic firmware setup involved in getting started with a Beta release of Lumorphix.

1.2.3.1 Setup

Ensure you have completed the previous setup stages (see Hardware (see page 7) and PC (see page 8)) and fulfilled the software requirements as per Beta Release Info (see page 98).

1.2.3.2 Binary

Select an appropriate (based on your target hardware and desired functionality) Lumorphix firmware binary from here[3].

1.2.3.3 Silicon

1.2.3.3.1 GOWIN

If your target hardware has GOWIN silicon, perform the following:

1. Configure the FPGA:
    a. If you are using Linux, via the following command:

```
$ openFPGALoader -b tangnano9k -f <Beta Release FPGA Firmware>.fs
```

    b. If on Windows, utilise the GOWIN programmer.
2. Verify the FPGA has been configured - you should notice the heartbeat LED is flashing.
3. Upload the Lumorphix Firmware
    a. Place the board into 'Firmware Update' mode - as detailed by the hardware-specific information that corresponds to your target hardware, for example here[4].
    b. Upload the corresponding Lumorphix Firmware via the following commands:
        i. If you are using Linux, (assuming no other USB devices connected - otherwise adjust 'USB**1**' accordingly) via the following commands:

```
$ sudo stty -F /dev/ttyUSB1 921600
$ sudo python firmware-uploader.py <Beta Release CPU Firmware>.v /
dev/ttyUSB1 921600
```

3 https://brisbanesilicon.com.au/products/lumorphix/beta-releases/
4 https://brisbanesilicon.scrollhelp.site/lumorphix/tang-nano-9k#TangNano9k-Userbutton

ii. If you are on Windows, determine which COM port the Tang Nano is connected to (via Device Manager), and use your Python IDE to run the following command (where <x> is the COM port number):

```
<IDE cmd prompt> firmware-uploader.py <Beta Release CPU
Firmware>.v 'COM<x>' 921600
```

4. Reset the Lumorphix Flash region (as it will contain rubbish after the initial power on) via the 'User button' section on page Tang Nano 9k (see page 101).

If you are planning to run the dual-core variant of Beta Release 02.0001[5] on the Tang Nano 9K you must perform the following steps:

1. Program the FPGA with the single-core variant of firmware
2. Upload the dual-core variant of CPU firmware
3. Program the FPGA with the dual-core variant of firmware

This is due to the fact that resource limitations prevent uploading CPU firmware with the single-core FPGA firmware.

1.2.3.3.2 AMD / Xilinx

If your target hardware has AMD / Xilinx silicon, perform the following:

1. Launch Vivado (or ML, Enterprise or Lab edition).
2. Open Hardware Manager
3. Select 'Open Target'
4. Select 'Add configuration memory device'
   a. Search and select the configuration memory part name listed in the corresponding child page of Hardware Targets (see page 100).
   b. Select 'OK'
   c. Navigate to the Lumorphix firmware binary in the 'Configuration file' selection field.
   d. Select 'OK'
5. Once the flash programming has completed, power cycle or hard reset (again see the corresponding child page of Hardware Targets (see page 100) on how to perform a hard reset) the board.
6. You should notice a heartbeat LED that is flashing to indicate the hardware has been successfully configured.

5 https://brisbanesilicon.com.au/products/lumorphix/beta-releases/#beta-release-02.0001

1.2.3.4 Next

See Connect! (see page 13) to setup a connection between your PC and Lumorphix.

1.2.4 Connect!

The following page details establishing a connection between your PC and Lumorphix.

1.2.4.1 Setup

To establish a connection between your PC and Lumorphix follow the steps outlined below.

1.2.4.1.1 Linux or Mac OS

1. Launch your preferred serial terminal emulator. See Serial Terminal Setup (see page 17) for more information.
2. Check your FTDI kernel module is loaded (on Linux it is built into the kernel, so is likely already loaded):

```
user@pc> lsmod | grep ftdi_sio
```

3. If not, load it:

```
user@pc> sudo modprobe -v ftdi_sio
```

4. If that failed, see here[6] for installation instructions (VCP driver).
5. A USB device should now be present in **/dev/** :

```
user@pc> ls /dev/ | grep USB1
ttyUSB1
```

6. Connect with baud=**921600** bps, data bits=8, stop bits=1, parity=none and flow control=none. The recommended tio[7] command line is as follows:

```
user@pc> sudo tio -b 921600 --output-delay 5 /dev/ttyUSB1
```

7. You're good to go! See section '**Test**' below, if you wish to test the connection.

6 https://ftdichip.com/document/installation-guides/
7 https://github.com/tio/tio

On some flavours of Linux, we've found that the device must be forced to baud 921600, prior to establishing a serial connection (step 6).

```
user@pc> stty -F /dev/ttyUSB1 921600
```

On some flavours of Mac OSX, we've found that the number of stop bits must be set to two, as part of establishing a serial connection (i.e. step 6).

```
user@pc> sudo tio -b 921600 --output-delay 5 /dev/ttyUSB1 -s 2
```

1.2.4.1.2 Windows

1. Launch 'Device Manager' (click 'Windows Start Button', start typing 'Device Manager').
2. Scroll down to 'Ports (COM & LPT), and expand it.
3. Note COM port number of entry 'USB Serial Port (COM<port number>). If there are multiple, unplug-plug Lumorphix to see which is the relevant port
4. Launch your serial client program (i.e. Putty).
5. Set 'Connection Type' to 'Serial', 'Serial line' to COM<Lumorphix port number> and 'Speed' to 921600.

If required, additional serial configuration is: data bits = 8, stop bits = 1, parity = none and flow control = none.

6. Open connection. You're good to go! See section '**Test**' below, if you wish to test the connection.

1.2.4.2 Test

After powering on the board and completing step outlined above in the '**Setup**' section, you are now connected to Lumorphix in its default mode (**REPL Mode**). In this mode, the user is presented with a Lua[8] **REPL** (Read-Eval-Print-Loop). For more information on this mode, see REPL Mode (see page 20).

To confirm this, simply press 'enter' on your keyboard, this will refresh the **REPL** prompt (it will print again, on a newline e.g. same as a Bash terminal, Python REPL etc.)

To review the boot log (which details various Lumorphix hardware and software configuration metadata) simply leave the connection open, and **hard reset** the board. The steps to perform a **hard reset** vary per hardware target - see the child page of Hardware Targets (see page 100) that corresponds to your current hardware.

For a detailed description of the boot log, see section '**Boot Log Description**' of page Boot Information (see page 18).

> Once you have established a serial connection to the board, you can print a help manual by simply typing '**help**' in **REPL Mode** or '**list|help**' in **Command Mode**.

1.2.4.3 Additional Setup

The only other software you will require, if you are going to develop and upload programs (as opposed to just use the **REPL**) is:

- The program upload script, available here[9].
- A Python installation.

1.2.4.3.1 Python installation

1.2.4.3.1.1 Linux or Mac OS

1. Launch your favourite terminal emulator.
2. Install Python.

```
user@pc> sudo apt install python3
```

8 https://www.lua.org/
9 https://brisbanesilicon.com.au/wp-content/uploads/2024/07/lumorphix_program_uploader.zip

3. Install PySerial.

```
user@pc> sudo pip install pyserial
```

1.2.4.3.1.2 Windows

1. The Python installer for windows is available here[10].
2. Ensure that you select 'Customize installation'-'Next' then tick 'Add Python to environment variables' and 'Precompile standard library'.

Ensure that you also install pyserial (after you've install Python, run the following command in Windows Powershell):

```
PS C:> pip install pyserial
```

Not all Beta releases support program upload - see Beta Release Info (see page 98) for more information.

10 https://www.python.org/downloads/windows/

1.3 Serial Terminal Setup

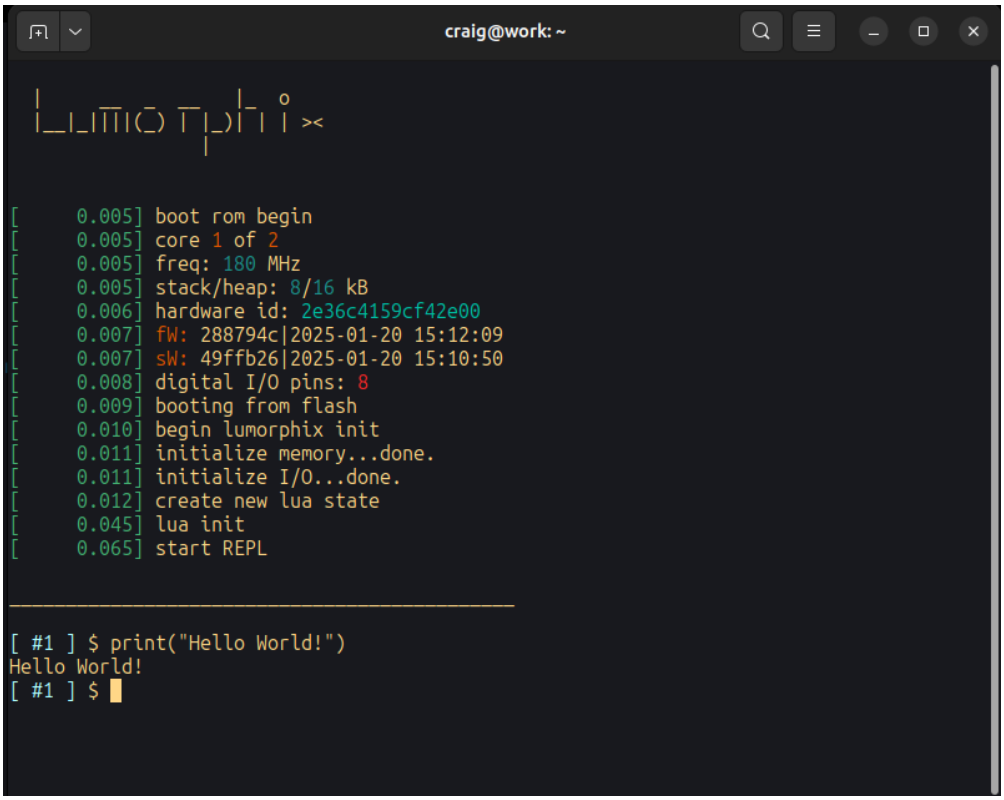
The following page details information related to the setup of the PC terminal used to communicate with Lumorphix.

1.3.1 Recommended Programs

Of the plethora of Serial Terminal emulators available, we recommend either **Tio**[11] or **Putty**[12].

1.3.2 Color Theme

A dark-style color theme is currently recommended, which will result in something similar to the following.



1 Dark Terminal Theme

11 https://github.com/tio/tio
12 https://www.putty.org/

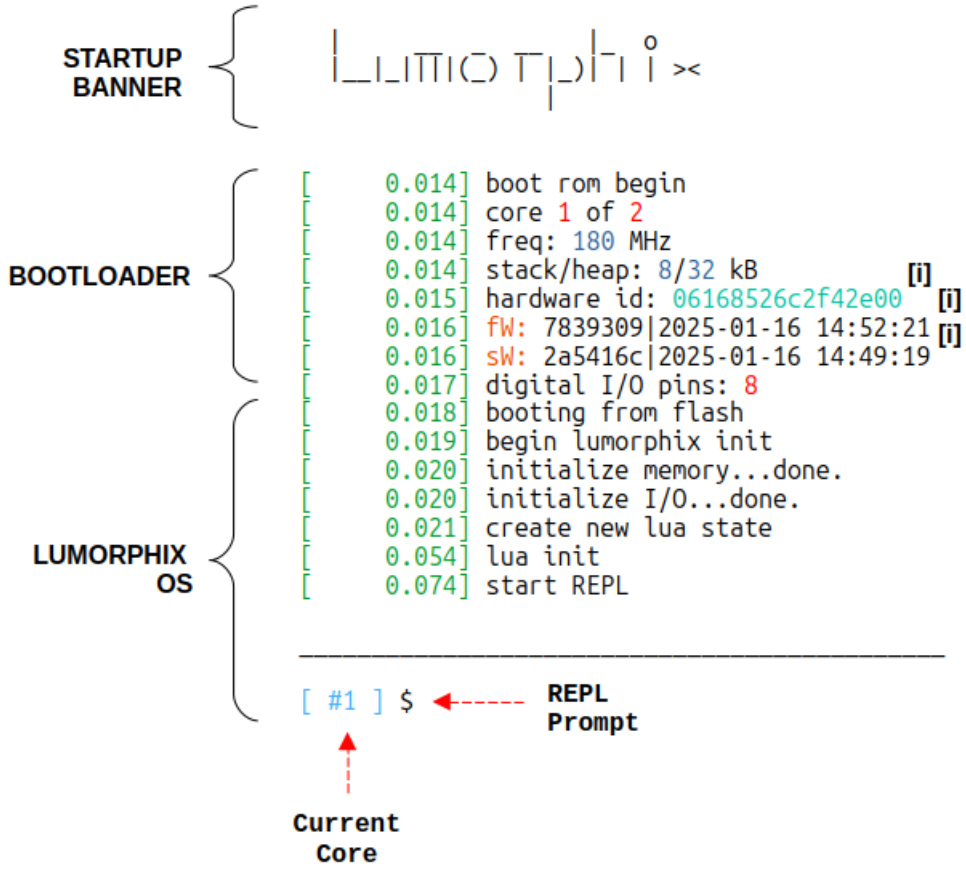
2 Boot Information

The following page details information related to the Lumorphix boot process.

2.1 Boot Log Description

Upon power-on, Lumorphix will output a boot log via the User Comms channel (for the Beta release, see Connect! (see page 13)).

An image of the boot log, followed by a description, is as follows (light terminal theme):



2 Lumorphix Boot Log

> There may be some slight differences between boot logs, depending upon the hardware and build configuration.

Boot Message	Description
core **x** of **y**	Lists current Core (**x**) and total Cores (**y**).
freq: **x** MHz	Lists the frequency (**x**) in MHz at which Lumorphix is clocked.
stack/heap: **x** / **y** kB	Lists the stack (**x**) and heap (**y**) sizes in kB with which the current Core was built.
hardware id: **x**	Lists the hardware DNA (**x**).
fW: **x** | **y**	Lists the firmware ID (**x**) and build date (**y**)
sW: **x** | **y**	Lists the software ID (**x**) and build date (**y**)
digital I/O pins: **x**	Lists the number of digital I/O pins (**x**) with which the current Core was built.

If the hardware DNA that Lumorphix was built with doesn't match the physical hardware DNA, then a warning message will be printed (if the Current Core is #1) and Lumorphix will only run for fifteen minutes.

2.1.1 References

[i] These log messages are only printed if the Current Core is #1.

3 REPL Mode

The following page details the default Lumorphix mode, the **REPL Mode**. If you are unfamiliar with a **REPL**, we recommend you familiarize yourself by reading some background information[13].

3.1 Prompt

The user is prompted by '**$**' in this mode, for example:

```
$
```

If Lumorphix has been configured for multiple Cores, by default the current Core number that the user is interacting with will precede the user prompt. For example, if the user is interacting with the fourth Core.

```
[ #4 ] $
```

3.2 Example

The following is a basic example of **REPL** usage.

```
$ foo = 123
$ foo
123
$ foo * foo
15129
$ foo_cubed = foo * foo * foo
$ foo_cubed
1860867
```

The above interaction with the Lumorphix **REPL** can be described as follows:

1. A variable named 'foo' is declared, and given the value 123.
2. The user typed 'foo' and pressed return - the Lumorphix **REPL** fetched that variable and printed its value.
3. The variable named 'foo' is multiplied by itself, which equates to 15129.

13 https://en.wikipedia.org/wiki/Read-eval-print_loop

4. A variable named 'foo_cubed' is declared, and given the value of foo * foo * foo.
5. Finally, the user typed 'foo_cubed' and pressed return - the Lumorphix **REPL** fetched that variable and printed its value.

One other feature of the REPL to note - multiline input is accepted. Every non-first line of a multiline script will be prompted by the the multiline prompt, '**$$**', until the end of the script section. For example:

```
$ if foo_cubed > 999 then
$$ print("Variable 'foo_cubed' is bigger than 999!")
$$ end
Variable 'foo_cubed' is bigger than 999!
```

3.3 User Interrupt

To prevent the critical failure case of remaining stuck in a scripted loop, or to simply exit a program or line of script earlier than it would naturally exit, simply press the '**q**' key on your keyboard. This is obviously a bit difficult to illustrate via an example, however consider the following interaction with the Lumorphix **REPL** on Core #2:

```
[ #2 ] $ import("sleep")
[ #2 ] $ while true do print("Forever ?"); sleep(1); end
Forever ?
Forever ?
Forever ?
Forever ?
stdin:1: <user interruption>
stack traceback:
    [C]: in function 'sleep'
    stdin:1: in main chunk
```

The user didn't want to remain stuck printing 'Forever ?' forever, so they pressed the keyboard '**q**' key after line 6 occurred.

3.4 History

Simply press the '**up**' or '**down**' arrow keys on your keyboard to access (up to) the previous ten entries. This is functionality is present for both the **REPL Mode** and **Command Mode**.

An entry wont be stored in history that exceeds one hundred and twenty eight characters, in order to minimise memory utilization.

3.5 Summary

The above information provides a brief overview of the Lumorphix **REPL**. Of course, in addition to supporting the Lua scripting language, Lumorphix offers a rich **API**, which allows interaction with various internal hardware components, as well as external I/O. See API (see page 32) for more information.

In additional to **REPL Mode**, Lumorphix also offers **Command Mode**, which allows the user to peruse and configure various hardware options (I/O, Core XBar, etc) and capabilities. See Command Mode (see page 23) for more information.

4 Command Mode

The following page details the Lumorphix **Command Mode**.

4.1 Activation

To activate command mode, from the **REPL**, simply type: **command** (short-form '**cmd**' is also accepted).

To deactivate command mode, simply type: **exit**.

4.2 Prompt

The user is prompted by '**/** ' in this mode. Again, if multiple Cores are enabled, the current Core number will precede the prompt - and the output will only be relevant to that Core. There is also a banner at the end of the command mode page. For example (three dots are essentially 'etc' - the command mode will occupy the entire Terminal window):

```
[ #1 ] /


...


COMMAND MODE | Type 'exit' to return to REPL | Type 'list|commands' to print a
short-list of commands | Type 'list|help' to print a detailed list of commands
```

4.3 Usage

4.3.1 Getting Help

To access help related information, two important commands are as follows:

Command	Description
list|commands	List supported commands.
list|help	List a summary of this documentation.

4.3.2 Supported Commands

A full list of supported commands, as well as a copy and description of their output is as follows.

4.3.2.1 List Commands

list|io_capabilities or **list|io_caps**

Lists the total I/O pins and capabilities of the current Core, for the current Lumorphix build. For example, the current Core has been built to support four I/O pins, and they have been built to support all I/O protocols apart from I2C.

```
DIGITAL I/O

       |GPIO_OUT|GPIO_IN |   PWM  |UART_OUT|UART_IN | SPI_OUT| SPI_IN |   I2C  ||SW_INTRPTS|HW_BUFFER|
  PIN1 |    X   |    X   |    X   |    X   |    X   |    X   |    X   |        ||     X    |   SML   |
  PIN2 |    X   |    X   |    X   |    X   |    X   |    X   |    X   |        ||     X    |   SML   |
  PIN3 |    X   |    X   |    X   |    X   |    X   |    X   |    X   |        ||     X    |   SML   |
  PIN4 |    X   |    X   |    X   |    X   |    X   |    X   |    X   |        ||     X    |   SML   |
```

list|io_type_cfg

Lists the current I/O configuration for each I/O pin, for the current Core. For example, the user has configured PIN1 as **PWM**, PIN2 as **GPIO_IN**, and PIN3 and PIN4 as **UART_OUT**.

```
I/O TYPE CONFIG

  PIN1        : PWM
  PIN2        : GPIO_IN
  PIN3        : UART_OUT
  PIN4        : UART_OUT
```

list|io_baud_cfg

Lists the baud configuration for each I/O pin that is configured as a **UART** (**IN** or **OUT**), for the current Core. For example, if the I/O type configuration remained the same as for the above command, the output of this command might be as follows (PIN3 and PIN4 are 9600 Baud).

```
I/O BAUD CONFIG

  PIN1        :   -
  PIN2        :   -
  PIN3        : 9600    |UART_OUT
  PIN4        : 9600    |UART_OUT
```

list|io_pwm_cfg

This command will list the oscillation frequency for each I/O pin that is configured as a **PWM**, for the current Core. For example, if the I/O type configuration remained the same as for the above command, the output of this command might be as follows (PIN1 oscillates at 10 KHz).

```
I/O PWM CONFIG

  PIN1        : 10 kHz    |PWM
  PIN2        :   -
  PIN3        :   -
  PIN4        :   -
```

list|io_spi_cfg

Lists the oscillation frequency for each I/O pin that is configured as a **SPI** (**IN** or **OUT**), for the current Core. For example, if PIN2 was configured as an **SPI_OUT**, the output of this command might be as follows (PIN3/4, whichever is configured as the **SPI_CLK**, would oscillate at 10 KHz).

```
I/O SPI CONFIG

  PIN1        :   -
  PIN2        : 10 kHz    |SPI_OUT
  PIN3        :   -
  PIN4        :   -
```

list|timer_cfg

Lists the timer build configuration for the current Core. It lists the enabled / disabled status of the following timers:

Timer	Description
General Timer	Enables any time-related API functions, for example '**sleep**'
Performance Timer	Allows the user to benchmark line(s) of script, or full programs. See the '**cycle|timeprompt**' command for further information.

If the 'General Timer' is disabled, and the user attempts to call a time-related API function, it will hang. For further information, see API (see page 32).

list|watchdog_cfg

Lists the watchdog build configuration for the current Core, including presence and timeout (if the watchdog is present, and the API function **watchdog_reset** is not called within the timeout period, the corresponding Core will reboot. See API (see page 32) for further information.

list|bus_cfg

Lists the presence of the FPGA fabric bus for the current Core.

list|xbar_cfg

Lists the Core cross-bar (xbar) build configuration. The cross-bar facilitates synchronization and communication between separate Cores. An example output, if the command was run on Core #1, for a Lumorphix built with four Cores, might be as follows:

```
XBAR LOCK

              |  -  |CORE2|CORE3|CORE4|
       LOCK EN|     |  X  |  X  |  X  |
    LOCK DEPTH|     |  1  |  1  |  1  |
     UNLOCK EN|     |     |     |     |
  UNLOCK DEPTH|     |     |     |     |


XBAR DATA

              |  -  |CORE2|CORE3|CORE4|
        TX EN|     |  X  |  X  |  X  |
 TX FIFO DEPTH|     |  1  |  1  |  1  |
        RX EN|     |  X  |  X  |  X  |
 RX FIFO DEPTH|     |  1  |  2  |  2  |
```

The locking mechanism from Core #1 to all other cores has been enabled, with a depth of one. The data transfer to/from Core #1 and all other cores has been enabled, with the return path between Core #3 and Core #1, and Core #4 and Core #1, having a FIFO of depth two. See API (see page 32) for more information.

list|user_comms_cfg

Lists the user communications (i.e. serial / terminal comms) config for the current Lumorphix build. Either internal (there are internal RTL modules per-Core within the IP that handles the user comms) or external (only a single user comms module, external to the IP). The internal option will have a larger LUT footprint, but also have the capability of communicating with each Core simultaneously.

list|clk_freq

Lists the clock frequency that Lumorphix is being clocked at, in MHz.

list|program_count

Lists the number of programs currently hosted on Lumorphix.

list|programs

Lists the programs currently hosted on Lumorphix.

list|start_on_boot_prompt_format

Lists the prompt format that has been configured to be loaded on boot (if any).

list|start_on_boot_program

Lists the program name that has been configured to start automatically on boot (if any).

list|program_code(“<Program Name>”)

Lists the program code of “<Program Name>”.

list|program_bytecode(“<Program Name>”)

Lists the program byte code of “<Program Name>”.

list|repl_history
list|cmd_history

Lists the **REPL** or **Command** history, up to a maximum of sixteen entries, each with a maximum length of sixty-four characters (longer input won’t be included).

4.3.2.2 Set Commands

set|io_type_cfg(PIN<x>, <I/O Type>)

Set PIN<x> (1 - maximum supported) to <I/O Type>. The I/O Type must be supported as per the output of the **list|io_caps** command.

set|io_baud_cfg(PIN<x>, <Baud Rate>)

Set PIN<x> (1 - maximum supported) that is configured as **UART_OUT** or **UART_IN** to have a baud rate of <Baud Rate>. Supported Baud Rates are as follows: 9600, 19200, 28800, 38400, 57600, 76800, 115200, 230400, 460800, 576000, 921600.

set|io_pwm_cfg(PIN<x>, <PWM Freq kHz>)

Set PIN<x> (1 - maximum supported) that is configured as **PWM** to have an oscillation frequency of <PWM Freq KHz>. Minimum 1 kHz, Maximum 200 Khz.

> At the upper range of PWM frequencies, there will be ranges that will be unachievable, depending upon the clock frequency. In this case, the frequency will be set to the closest achievable that is larger than the requested frequency.

set|io_spi_cfg(PIN<x>, <SPI Freq kHz>)

Set PIN<x> (1 - maximum supported) that is configured as **SPI** (**IN** or **OUT**) to have an oscillation frequency of <SPI Freq KHz>. Minimum 1 kHz, Maximum 250 Khz.

set|start_on_boot_io_type_cfg

Set the current I/O type configuration to be the start-on-boot default.

set|start_on_boot_prompt_format

Set the current prompt format to be the start-on-boot default.

set|start_on_boot_program("<Program Name>")

Set <Program Name> to be the start-on-boot default.

4.3.2.3 Reset Commands

reset|start_on_boot_prompt_format

Reset the prompt format that has been configured to be loaded on boot to 'Unconfigured'.

reset|start_on_boot_io_type_cfg

Reset the I/O type configuration to be loaded on boot to 'None'.

reset|start_on_boot_program

No program will start on boot.

reset|io_type_cfg(PIN<x>)

Reset PIN<x> (1 - maximum supported) to Type 'None'.

reset|all_io_type_cfg

Reset all PINs to Type 'None'.

4.3.2.4 Delete Commands

delete|program("<Program Name>")

Delete the program "<Program Name>".

delete|all_programs

Delete all programs.

4.3.2.5 Load Commands

load|start_on_boot_io_type_cfg

Load the start-on-boot I/O type configuration.

4.3.2.6 Upload Commands

upload|program("<Program Name>")

Upload a program "<Program Name>" to the device. Requires the program uploader[14] utility. After running the command, disconnect your terminal program and upload the program from your PC, for example, on Linux (assuming no other USB devices connected - otherwise adjust 'USB**1**' accordingly):

```
$ python program_uploader.py <program name>.lua /dev/ttyUSB1 921600
```

Windows (run within your Python IDE and determine the COM port number (<x>) of your device via Device Manager):

```
<IDE cmd prompt> program_uploader.py <program name>.lua COM<x> 921600
```

4.3.2.7 Run Commands

run|program("<Program Name>")

Run the program "<Program Name>".

> This command, if successful, will return the user to REPL mode after the program has completed.

14 https://brisbanesilicon.com.au/utilities/program-uploader.py

run|reboot

Reboot the current Core.

4.3.2.8 Cycle Commands

cycle|cpuprompt

Cycle the CPU prompt between unconfigured, on or off.

cycle|timeprompt

Cycle the time prompt between the following:

Prompt Color	Description
N/A	No prompt.
Green	Absolute time since power-on.
Purple	Load (compile) and call duration (**REPL** or program).
Cyan	Call duration only (**REPL** or program).
Red	Load (compile) duration only (**REPL** or program).

4.3.2.9 Memory Commands

memory|free

Display the current memory that is free, in bytes.

memory|total

Display the total memory that the current Core was built with, in bytes.

memory|low_water_mark

Display the minimum amount of memory that was ever free, in bytes.

memory|reset_low_water_mark

Reset the minimum amount of memory that was ever free to the current amount of memory that is free.

4.3.2.10 Stack Commands

stack|total

Display the total stack size that the current Core was built with, in bytes.

stack|low_water_mark

Display the minimum amount of stack memory that was ever free, in bytes.

stack|reset_low_water_mark

Reset the minimum amount of stack memory that was ever free to the current amount of stack memory that is free.

4.3.2.11 Other Commands

exit

Return to **REPL** mode.

5 API

The following section details the Lumorphix API. The API can be split (roughly) into a set of in-built constants, and a set of library functions that utilise them, and any other Lua supported constructs. This section first details the constants followed by the library functions.

5.1 Constants

All library constants are integers e.g. you can print the library constants value by entering its name into the REPL. Note that all library constants are immutable - any attempt to modify them will have no effect. The library constants are detailed as follows:

5.1.1 Core

Constant	Description

CORE<x>	Represents CORE number 'x', For example, in the REPL (or an uploaded program):
	`$ print("Value of CORE3 is: "..CORE3)`
	`Value of CORE3 is: 3`

5.1.2 Pin

Constant	Description

PIN<x>	Represents I/O PIN number 'x', For example, in the REPL (or an uploaded program):
	`$ print("Value of PIN7 is: "..PIN7)`
	`Value of PIN7 is: 7`

PIN<x>_ BITMASK	Represents I/O PIN number 'x' bitmask. For example, handling interrupts:
	`$ function interrupt(interrupt_vector)`
	`  if (interrupt_vector & PIN3_BITMASK) ~= 0 then`
	`  print("Interrupt occurred on PIN3!")`
	`  end`
	`end`

5.1.3 I/O Level

Constant	Description
LOW	Represents a voltage low (0 volts) signal level.
HIGH	Represents a voltage high (3.3 volts) signal level.
TOGGLE	Represents an inverse voltage signal level, i.e. high the opposite of low. For example: `$ set_gpio(PIN7, TOGGLE)`

5.1.4 I/O Type

Constant	Description
NONE	Represents an I/O type of none.
GPIO_OUT	Represents an General Purpose, 'Digital I/O' type with direction output.
GPIO_IN	Represents an General Purpose, 'Digital I/O' type with direction input.
PWM	Represents a Pulse-Width-Modulation, 'Digital I/O' type.
UART_OUT	Represents a UART communications protocol, 'Digital I/O' pin type, with direction output.
UART_IN	Represents a UART communications protocol, 'Digital I/O' pin type, with direction input.
SPI_OUT	Represents a SPI communications protocol, 'Digital I/O' pin type, with direction output.
SPI_IN	Represents a SPI communications protocol, 'Digital I/O' pin type, with direction input.

I2C	Represents a I2C communications protocol, 'Digital I/O' pin type, with direction input and output. For example: `$ set_io_config(PIN14, I2C)`

5.1.5 GPIO Interrupt

Constant	Description
GPIO_ INTRPT_ LOW	Represents a low (0 volts) interrupt type on an I/O configured as GPIO_IN.
GPIO_ INTRPT_ HIGH	Represents a high (3.3 volts) interrupt type on an I/O configured as GPIO_IN.
GPIO_ INTRPT_ RISING_ EDGE	Represents a low to high transition interrupt type on an I/O configured as GPIO_IN.

GPIO_ INTRPT_ FALLING_ EDGE	Represents a high to low interrupt type on an I/O configured as GPIO_IN. For example, parsing interrupt types:
	`$ gpio_interrupt_vector = get_interrupts_on_pin(PIN6)`
	`if (tgpio_interrupt_vector & GPIO_INTRPT_RISING_EDGE) ~= 0`
	`then`
	`  print("Detected rising edge on PIN6!")`
	`end`

5.1.6 UART Interrupt

Constant	Description

UART_RX_ INTRPT_ DATA_ AVAILABLE	Represents data available interrupt type on an I/O configured as UART_IN.

5.2 Functions

5.2.1 Base Library

The library functions of the base library are detailed by the following table. Make sure you read the purple 'Note' at the bottom of this page, regarding the '**import**' function.

5.2.1.1 Import

Function	Description
import	Import function from base library (ARG1: <base library function>). Import function from non-base library (ARG1: <library>, ARG2: <library function>). For example: `$ import("set_io_type_cfg")`

5.2.1.2 I/O Config

Function	Description
set_io_type_cfg	Configure I/O (ARG1: PIN<x>) as type (ARG2: NONE, GPIO_OUT, GPIO_IN, PWM, UART_OUT, UART_IN, SPI_OUT, SPI_IN or I2C).
reset_io_type_cfg	Reset I/O (ARG1: PIN<x>) to type NONE.
reset_all_io_type_cfg	Reset all I/O to type NONE (no arguments). For example: `$ reset_all_io_type_cfg()`

5.2.1.3 GPIO

Function	Description
set_gpio	Set I/O (ARG1: PIN<x>), configured as GPIO_OUT, to level (ARG2: LOW, HIGH or TOGGLE). For example: `$ set_gpio(PIN7, HIGH)`

get_gpio	Get I/O (ARG1: PIN<x>), configured as GPIO_IN, level. For example:
	`$ my_kite = get_gpio(PIN2)`
	`$ if my_kite == HIGH then print(" my_kite is HIGH!")`
	`$$ else print("my_kite is not HIGH!") end`
	`my_kite is HIGH!`

5.2.1.4 PWM

Function	Description
set_pwm	Set I/O (ARG1: PIN<x>), configured as PWM, to duty cycle (ARG2: **0** - **PWM_MAX**). The value of '**PWM_MAX**' is 256 (100% duty).

5.2.1.5 SPI

Function	Description

spi_tx	Same as 'spi_tx_byte'. The behavior in the case the ARG1 I/O PIN is configured to have a non byte-multiple data width: • If the data width, X, is less than one byte, each byte that is sent to the SPI module will have the most-significant [8 - X] bits discarded, the remainder are transmitted. • If the data width, X, is greater than one byte, each byte that is sent to the SPI module is buffered until a byte-multiple of data is stored. Then the most-significant [Buffered-Bits - X] bits are discarded, the remainder are transmitted.
spi_tx_byte	Send 1 byte (ARG2: integer) of data to the I/O (ARG1: PIN<x>), configured as **SPI_OUT**. It is possible this will NOT result in an actual SPI transmission (if it is less than the build configured number of bits per SPI transmission). See description of '**spi_tx**'.
spi_tx_char	Send 1 byte (ARG2: string) of data to I/O (ARG1: PIN<x>), configured as **SPI_OUT**.
spi_tx_int	Send 4 bytes (ARG2: integer) of data to I/O (ARG1: PIN<x>), configured as **SPI_OUT**. It is possible this will result in multiple SPI transmissions.
spi_rx_byte	Receive one byte from I/O (ARG1: PIN<x>), configured as **SPI_IN**. This function will hang if no SPI data has been received.
spi_rx_byte_ nonblocking	Not included in Beta Release.

spi_rx_char	Receive one character from I/O (ARG1: PIN<x>), configured as **SPI_IN**. This function will hang if no SPI data has been received.
spi_rx_char_ nonblocking	⚠ Not included in Beta Release.

5.2.1.6 UART

Function	Description
uart_tx	Same as '**uart_tx_byte**'. For example: `$ uart_tx(PIN4, 0x30)`
uart_tx_char	Send 1 byte (ARG2: string) of data to the I/O (ARG1: PIN<x>), configured as **UART_OUT**.
uart_tx_byte	Send 1 byte (ARG2: integer) of data to the I/O (ARG1: PIN<x>), configured as **UART_OUT**.
uart_tx_int	Send 4 bytes (ARG2: integer) of data to the I/O (ARG1: PIN<x>), configured as **UART_OUT**.
uart_rx_byte	Receive and return 1 byte (as an integer) of data from the I/O (ARG1: PIN<x>), configured as **UART_IN**. This function will block if the corresponding UART module currently has no data, until a byte of data is received.

uart_rx_byte_ nonblocking	Receive and return 1 byte (as an integer) of data and return 'byte valid' from the I/O (ARG1: PIN<x>), configured as **UART_IN**. This function will not block - if the corresponding **UART_IN** module currently has no data then 'byte valid' will be false.
uart_rx_char	Same as '**uart_rx_byte**', however the returned UART data will be a character.
uart_rx_char_ nonblocking	Again, same as '**uart_rx_byte_nonblocking**', however the returned UART data will be a character.

5.2.1.7 I2C

Function	Description
i2c_tx	Not included in Beta Release.
i2c_tx_char	Not included in Beta Release.
i2c_tx_byte	Not included in Beta Release.
i2c_tx_int	Not included in Beta Release.

i2c_rx_byte	⚠ Not included in Beta Release.
i2c_rx_byte_ nonblocking	⚠ Not included in Beta Release.
i2c_rx_char	⚠ Not included in Beta Release.
i2c_rx_char_ nonblocking	⚠ Not included in Beta Release.

5.2.1.8 FPGA Bus

Function	Description
fpga_write	Write a integer (ARG1: integer) to the FPGA bus. Block until the FPGA bus 'ready' strobe is asserted.
fpga_write_nonbloc king	Attempt to write a integer (ARG1: integer) to the FPGA bus. Doesn't block, returns true / false to indicate if the write was successful (i.e. the FPGA bus 'ready' strobe was asserted).
fpga_read	Read a integer from the FPGA bus. Block until the FPGA bus 'ready' strobe is asserted.
fpga_read_nonblock ing	Read a integer from the FPGA bus. Doesn't block, returns the value read (0 in invalid) and true / false to indicate if the read was successful (i.e. the FPGA bus 'ready' strobe was asserted).

5.2.1.9 Interrupt

Function	Description
global_ interrupt_ enable	Global interrupt enable. The 'interrupt' function will trigger if an enabled interrupt to occurs (no arguments).
global_ interrupt_ disable	Global interrupt disable. The 'interrupt' function will not trigger even if an enabled interrupt to occurs (no arguments).
repl_ interrupt_ mode_enable	This interrupt mode is enabled by default prior to the execution of each new atomic (i.e. single or set of multi-line) **REPL** input. It is disabled by default prior to the execution of a program. Thus only a multi-line **REPL** input, which calls '**repl_interrupt_mode_disable**' will proceed with this mode disabled, and only a program which calls '**repl_interrupt_mode_enable**' will proceed with this mode enabled (from the point at which it is enabled / disabled, of course). If enabled, any individual interrupt source that is enabled and active at the end of the entire **REPL** input will trigger an interrupt at that point (i.e. just prior to returning to user prompt). This is in addition to the typical interrupt functionality of triggering between each line of script (i.e. in the case of a **REPL**, a multi-line **REPL** input). Also, if the interrupts have been enabled globally (i.e. '**global_interrupt_enable**') this the global interrupt enable will persist across individual, atomic calls to the **REPL** (i.e. across single-line calls, or separate multi-line calls). Otherwise, the global interrupt configuration will be reset to disabled at the beginning of each new input into the **REPL** (i.e. a single-line or multi-line input has completed, and the user has been prompted to enter new Lua script).
repl_ interrupt_ mode_disable	See above description.

set_interrupt_types_for_pin	Enable only specified interrupts by bitmask (ARG2: bit-wise combination of <PIN<x> Type>_INTRPT_<type>) for input I/O (ARG1: PIN<x>). For example:
	`$ set_io_config(PIN1, GPIO_IN)`
	`$ set_interrupt_types_for_pin(PIN1, GPIO_INTRPT_LOW | GPIO_INTRPT_RISING_EDGE | GPIO_INTRPT_FALLING_EDGE)`
	`$ print("Enabled Interrupt Types of 'Low' 'Rising Edge' and 'Falling Edge' for GPIO_IN PIN1")`
	`Enabled Interrupt Types of 'Low' 'Rising Edge' and 'Falling Edge' for GPIO_IN PIN1`

enable_interrupt_types_for_pin	Same as 'set_interrupt_types_for_pin' however additionally enable specified interrupts, as well as whatever is already enabled.
disable_interrupt_types_for_pin	Disable specified interrupts by bitmask (ARG2: bit-wise combination of <PIN<x> Type>_INTRPT_<type>) for input I/O (ARG1: PIN<x>).
get_interrupts_on_pin	Get interrupt vector for I/O (ARG1: PIN<x>).
ack_interrupt_types_on_pin	Acknowledge specified interrupts by bitmask (ARG2: bit-wise combination of <PIN<x> Type>_INTRPT_<type>) for I/O (ARG1: PIN<x>).
ack_interrupt_types_on_pins	Acknowledge specified interrupts by bitmask (ARG2: bit-wise combination of <PIN<x> Type>_INTRPT_<type>) for I/O specified by bitmask (ARG1: PIN<x>_BITMASK).

interrupt_handler	Users redefine this function with a single argument, 'interrupt_vector', and it will be called upon an enabled interrupt occurring. When called, the 'interrupt_vector' argument will reflect all PIN<x> upon which an interrupt occurred, via the corresponding 'PIN<x>_BITMASK'. For example:
	
	`$ set_interrupt_types_for_pin(PIN4, GPIO_INTRPT_LOW | GPIO_INTRPT_RISING_EDGE)`
	`$ set_interrupt_types_for_pin(PIN6, GPIO_INTRPT_FALLING_EDGE)`
	`$ global_interrupt_enable()`
	`$ function interrupt_handler(interrupt_vector)`
	`  if (interrupt_vector & PIN4_BITMASK) ~= 0 then`
	`  print("Interrupt occurred on PIN4!")`
	`  end`
	`  if (interrupt_vector & PIN6_BITMASK) ~= 0 then`
	`  print("Interrupt occurred on PIN6!")`
	`  end`
	`end`
	
	`interrupt.bind(interrupt_handler)`

5.2.1.10 Sleep

Function	Description
sleep	Pause for X (ARG1: integer) seconds. Interruptible.
sleep_f	Pause for X (ARG1: float) seconds. Interruptible.
msleep	Pause for X (ARG1: integer) milliseconds. Interruptible.
msleep_f	Pause for X (ARG1: float) milliseconds. Interruptible.
usleep	Pause for X (ARG1: integer) microseconds. Interruptible.
sleep_noint	Pause for X (ARG1: integer) seconds. Uninterruptible.
mssleep_noint	Pause for X (ARG1: integer) milliseconds. Uninterruptible.
usleep_noint	Pause for X (ARG1: integer) microseconds. Uninterruptible.

5.2.1.11 Watchdog

Function	Description
watchdog_reset	Reset the hardware watchdog back to its start timer value. See Command Mode (see page 23) for more information. If the watchdog is not present, calling this function will have no effect.
get_watchdog_timer	Get the current watchdog timer value. See Command Mode (see page 23) for more information. If the watchdog is not present, calling this function will return zero.

5.2.1.12 Core Communication

Function	Description

pipe_tx_byte	Send 1 byte (ARG2: integer) of data to CORE (ARG1: PIN<x>), blocking if the channel is currently busy (i.e. the FIFO is **full**). Attempts to send data to self will throw an error. The xbar data route must be enabled in the build configuration of the XBar, or the function will hang - see Command Mode (see page 23) for more information. For example:
	`[ #1 ] $ pipe_tx_byte(CORE2, 0xde)`
	`[ #1 ] $ pipe_tx_byte(CORE3, 0xad)`
	`[ #1 ] $ pipe_tx_byte(CORE4, 0xbe)`
	`[ #1 ] $ pipe_tx_byte(CORE1, 0xef)`
	`stdin:1: bad argument #1 to 'pipe_tx_byte'`
	`(invalid core number)`
	`stack traceback:`
	`[C]: in function 'pipe_tx_byte'`
	`stdin:1: in main chunk`

pipe_tx_byte_nonblocking	Attempt to send 1 byte (ARG2: integer) of data to CORE (ARG1: PIN<x>), will not block if the channel is currently busy (i.e. the xbar data FIFO is full). Returns an integer representing the number of bytes sent. The route must be enabled in the build configuration of the XBar, or the function will hang.

pipe_rx_byte	Receive 1 byte (ARG2: integer) of data from CORE (ARG1: PIN<x>), blocking if the channel is currently busy (i.e. the FIFO is **empty**). Attempts to send data to self will throw an error. The xbar data route must be enabled in the build configuration of the XBar, or the function will hang.
pipe_rx_byte_ nonblocking	Attempt to receive 1 byte (ARG2: integer) of data from CORE (ARG1: PIN<x>), will not block if the channel is currently busy (i.e. the xbar data FIFO is full). Returns an integer representing the number of bytes received, and the actual byte received (zero if none received). The route must be enabled in the build configuration of the XBar, or the function will hang.

5.2.1.13 Core Synchronization

Function	Description
lock	Lock xbar barrier between Core<self> and Core <x> (ARG1: integer). Will block until barrier is unlocked by Core<x>. On power-on, all xbar barriers will begin in the 'unlocked' state, to the depth listed in the Xbar config (see Command Mode (see page 23)).
lock_ nonblocking	Attempt to lock xbar barrier between Core<self> and Core <x> (ARG1: integer). Will not block if barrier has not been unlocked by that Core. Returns '1' to represent successful locking of the barrier, '0' otherwise.
unlock	Unlock xbar barrier between Core<self> and Core <x> (ARG1: integer). Will block until barrier is locked by Core<x>.
unlock_ nonblocking	Attempt to unlock xbar barrier between Core<self> and Core <x> (ARG1: integer). Will not block if barrier has not been locked by Core<x>. Returns '1' to represent successful unlocking of the barrier, '0' otherwise.

5.2.1.14 Program

Function	Description

run_program	Run the program named (ARG1: string). For example:
	`$  cmd`
	`/ list|programs`
	`  Program 1.  : hello_world.lua`
	`  Program 2.  : the_day_my_bum_went_psycho.lua`
	`  Program 3.  : gpio_example.lua`
	`/ exit`
	`$ run_program("the_day_my_bum_went_psycho.lua")`
	`This is the story of the day my Bum went Psycho.`
	`..actually forget it. Program end.`

5.2.1.15 Power

Function	Description
reboot	Reboot the current Core.
exit	Exit the current program or statement (**REPL**), , irrespective of nested program level. If provided, returns argument 1, of any type (ARG1: obj).

By default, all functions except '**reboot**', '**exit**' and '**import**' need to be first imported prior to being called. This is by design, in order to reduce memory usage.

It is possible to import the entire API in a single call to '**import("all")**.'

5.2.2 Interrupt Library

Function	Description
bind	Bind the interrupt handler (ARG1: function) to the global interrupt source. Is part of the interrupt library, so must be called as such: `$ interrupt.bind(interrupt_handler)`
call	Explicitly call the interrupt handler that is bound to the global interrupt source. Is part of the interrupt library, so must be called as such: `$ interrupt.call()`

5.2.3 Third-party Libraries

There is some support for the following built-in Lua libraries. Use the '**import**' function, with the library name as the first argument, to import them.

- string[15]
- table[16]
- math[17]

15 https://www.lua.org/pil/20.html
16 https://www.lua.org/pil/19.html
17 https://www.lua.org/pil/18.html

5.3 Variables

There are a smattering of variables that are setup as part of the boot process. They are described as follows:

Variable	Description
_VERSION	Lumorphix Lua version.
_sW	Software build ID and date (same as output by boot log).
_fW	Firmware build ID and date (same as output by boot log).
_hW	Hardware DNA (same as output by boot log).
_cI	Current Core number and total Cores.
PWM_MAX	Maximum PWM duty cycle value (256).

6 Example Usage

The following section details examples of common usage of Lumorphix.

Many of the following examples utilise the **REPL** - however those examples omit the **REPL** prompt (**$**) in order to allow the example script to be copy-paste by the user. If the **Command Mode** is part of an example, it will always explicitly include the '**/**' prompt.

6.1 Toggle GPIO

The following page provides an example of toggling a GPIO.

6.1.1 Setup

So you've booted Lumorphix and connected to the **REPL** as per (if a Beta release) Connect! (see page 13). You now wish to achieve the 'Hello World' of embedded systems, and toggle a GPIO. Simply perform the following:

```
import("all")
set_io_type_cfg(PIN1, GPIO_OUT)
set_gpio(PIN1, HIGH)
```

The PIN1 output should now be high, i.e. VCC (3.3 Volts). That's all there is to it!

Instead of importing the entire Lumorphix API, the import script section could instead be as follows (the following would minimise memory footprint):

```
import("set_io_type_cfg")
import("set_gpio")
```

To set it low again:

```
set_gpio(PIN1, LOW)
```

To toggle it:

```
set_gpio(PIN1, TOGGLE)
```

If you are running Lumorphix as a Beta release, the value of PIN1 (and possibly PIN2, PIN3 - depending upon the Beta Release) will be reflected by the state of an LED. Thus you should notice it turn on and off with the above commands. See the child page of Hardware Targets (see page 100) that corresponds to your target hardware as to which LED(s) reflect which PIN state.

6.1.2 Additional Info

After running the above script, you could review the PIN configuration (below output would be if Core #1 was built with eight I/O):

```
[ #1 ] $ cmd
/ list|io_type_cfg

  I/O TYPE CONFIG

    PIN1    : GPIO_OUT
    PIN2    : NONE
    PIN3    : NONE
    PIN4    : NONE
    PIN5    : NONE
    PIN6    : NONE
    PIN7    : NONE
    PIN8    : NONE
```

6.1.3 References

Connect! (see page 13)

API (see page 32)

Command Mode (see page 23)

6.2 UART Transmit

The following page provides an example of transmitting a string via UART.

6.2.1 Example

The first step is, as usual, configuring the PIN to the desired I/O type, in this case **UART_OUT**. First the type is reset (this has no effect if it is already of type **NONE**), then it is set as the desired type. Just to be slightly different from the other examples, we will use PIN4 as the UART I/O.

```
import("reset_io_type_cfg")
import("set_io_type_cfg")

reset_io_type_cfg(PIN4)
set_io_type_cfg(PIN4, UART_OUT)
```

To confirm that PIN4 has been configured as **UART_OUT**, one can use the '**list|io_type_cfg**' command when in **Command Mode**.

The next and final step of the example is to perform the string transmission.

```
import("uart_tx_char")
import("string", "sub")

str = "Hello World"
for i=1,#str do
uart_tx_char(PIN4, string.sub(str, i, i))
end
```

6.2.2 References

Command Mode (see page 23)

API (see page 32)

Lua String Library[18]

18 https://www.lua.org/pil/20.html

6.3 SPI Receive

The following page provides an example of receiving data via SPI.

6.3.1 Example

The I/O PIN is first configured to the desired I/O type, in this case **SPI_IN**. When using protocols that require more than one PIN, the subsequent PIN(s) (in this case two - **CS** and **CLK**) are used, and must **not** be currently configured as some other I/O type (or an error will occur) - they must be of type **NONE**.

```
import("all")
set_io_type_cfg(PIN2, SPI_IN)
```

To confirm that PIN2, PIN3 and PIN4 have been configured as **SPI_IN**, **SPI_IN_CLK** and **SPI_IN_CS** one can use the '**list|io_type_cfg**' command when in **Command Mode**. In this case '**SPI_IN**' is the data PIN, the other two are obvious to anyone familiar with the spi protocol[19].

The next and final step of the example is to perform the spi receive. We don't want to hang until the actual data is received, so we use the non-blocking function.

```
byte, got_byte = spi_rx_byte_nonblocking(PIN2)
print("\tbyte: "..byte.."\n")
  byte: 0

print("got_byte: "..got_byte.."\n")
  got_byte: false

sleep(10)
byte, got_byte = spi_rx_byte_nonblocking(PIN2)
print("byte: "..byte.."\n")
  byte: 0xde

print("got_byte: "..got_byte)
  got_byte: true
```

Nothing was received in the example above at first, then one byte was!

6.3.2 References

Command Mode (see page 23)

API (see page 32)

19 https://en.wikipedia.org/wiki/Serial_Peripheral_Interface

6.4 LED Fade

The following page provides an example of fading an LED in and out.

6.4.1 Background

To modify the brightness of an LED, we utilise Pulse-Width-Modulation[20] (PWM) - modifying the duty cycle of a square wave to control its average supply power.

6.4.2 Example

As usual, first the required functions from the API are imported.

```
import("reset_all_io_type_cfg")
import("set_io_type_cfg")
import("set_pwm")
import("set_gpio")
import("msleep_f")
```

Next (again, as usual) set the I/O to the desired type:

```
PWM_PIN = 1
LOOP_COUNT = 10

reset_all_io_type_cfg()
set_io_type_cfg(PWM_PIN, PWM)
set_io_type_cfg(PIN2, GPIO_OUT)
set_io_type_cfg(PIN3, GPIO_OUT)
```

A helper function, which returns true if the number argument passed to it is even (false otherwise), is added to simplify some of the script:

```
function is_even(x)
    return ((x & 1) == 0)
end
```

Set other LEDs (present depending upon target hardware) to GND in order to not obscure the fading LED.

20 https://en.wikipedia.org/wiki/Pulse-width_modulation

```
-- Set other LEDs to GND

set_gpio(PIN2, LOW)
set_gpio(PIN3, LOW)
```

The core LED fade algorithm is as follows:

```
for i=1,LOOP_COUNT do
    for j=5,PWM_MAX do
        if is_even(i) then
            set_pwm(PWM_PIN, j)
            msleep_f(100/j)
        else
            set_pwm(PWM_PIN, (PWM_MAX-j))
            msleep_f(100/((PWM_MAX+5)-j))
        end
    end
end
```

On many of the Beta release hardware boards, PIN1 is also tied to LED4. If running on a Beta release board, you will notice LED4 fade in and out (so don't require external hardware to achieve this).

Finally, lets discard all of the variables, functions etc, that are now unused (to re-acquire the memory they utilise). Also, for good measure, force a garbage collection.

```
PWM_PIN = nil
LOOP_COUNT = nil

reset_all_io_type_cfg = nil
set_io_type_cfg = nil
set_pwm = nil
msleep_f = nil

collectgarbage()
```

That's it!

6.4.3 References

API (see page 32)

Command Mode (see page 23)

6.5 Fibonacci Sequence Calculation

The following page provides an example of calculating the Fibonacci Sequence, and performance profiling it.

6.5.1 Setup

Since we want to performance profile the calculation, we first enable the performance timer, of 'program execution only'.

```
$ cmd
/ cycle|timeprompt
[    15.951] / cycle|timeprompt
[     0.000] / cycle|timeprompt
[     0.000] / exit
[     0.000] $
```

After running the above, the time prompt should be a light blue / cyan color (this indicates the performance timer of ‘program execution only’).

6.5.2 Script

Lets first define the Fibonacci function, naming it accordingly.

The prompt has been omitted from the following snippet in order to allow the user to copy-paste it with ease.

```
function FiboNonRecursive(n)
     a = 0
     b = 1
     c = (n >> 1)
     for i=1,c do
         a=a+b
         b=a+b
     end
     if (n & 1) == 1 then
         return b;
     else
         return a;
```

```
    end
end
```

Next, lets call it!

```
[     0.000] $ FiboNonRecursive(35)
9227465
[ #1 ] [     0.004] $
```

6.5.3 Summary

So the **35th** series of the Fibonacci sequence is equal to **9227465** was calculated in **4 milliseconds**.

References

Command Mode (see page 23)

fibonacci_nonrecursive.lua (see page 84)

6.6 Float Divide and String Concatenation

The following page provides an example of peforming 100x floating point division, concatenating the result with a string, and printing it.

6.6.1 Example

```
limit = 111111111;
for i = 1,100 do
    foo = math.random(limit)
    foo_div = foo / 1.12345
    print("foo: "..foo.."\tfoo_div: "..foo_div)
end
```

The output of running the above is quite long and is mostly numeric. Run it yourself if you wish to peruse it!

6.7 Interrupt

The following page provides an example of setting up and triggering an interrupt.

6.7.1 Hardware Setup

For this example to function, connect a wire or resistor between PIN1 and PIN2.

6.7.2 Setup

For this example, we will just import the entire Lumorphix library:

```
import("all")
```

Next we initialise the GPIO:

```
reset_all_io_type_cfg()
set_io_type_cfg(PIN1, GPIO_OUT)
set_io_type_cfg(PIN2, GPIO_IN)
```

Next we reset the interrupt functionality.

```
global_interrupt_disable()
set_interrupt_types_for_pin(PIN2, NONE)
ack_interrupt_types_on_pin(PIN2, 0xFF)
    -- NOTE: acknowledge any existing interrupts
```

Defining an interrupt handler is next.

```
function interrupt_handler(interrupt_vector)
    if (interrupt_vector & PIN2_BITMASK) ~= 0 then
        gpio_interrupt_vector = get_interrupts_on_pin(PIN2)
        ack_interrupt_types_on_pin(PIN2, gpio_interrupt_vector)
        if (gpio_interrupt_vector & GPIO_INTRPT_RISING_EDGE) ~= 0 then
            print("\tRising edge!")
        end
        if (gpio_interrupt_vector & GPIO_INTRPT_FALLING_EDGE) ~= 0 then
            print("\tFalling edge!")
       end
    end
```

```
end
```

Binding the defined interrupt handler to the global interrupt source is the next, critical, step.

```
interrupt.bind(interrupt_handler)
```

When you are ready to start triggering interrupts, enable the global interrupt source, and the desired interrupt edge(s).

```
global_interrupt_enable()
set_interrupt_types_for_pin(PIN2, GPIO_INTRPT_FALLING_EDGE | GPIO_INTRPT_RISING_EDGE)
```

Once this is complete, one can trigger some interrupts!

```
set_gpio(PIN1, HIGH)
    Rising edge!
set_gpio(PIN1, LOW)
    Falling edge!
set_gpio(PIN1, HIGH)
    Rising edge!
set_gpio(PIN1, LOW)
    Falling edge!
```

6.7.3 References

API (see page 32)

gpio_interrupt.lua (see page 77)

6.8 Inter-Core Communication

The following page provides an example of transferring data between heterogeneous Cores.

6.8.1 Setup

This example requires Lumorphix to be built to support more than one Core, and to have the data channel (XBar Data) between at least two of them enabled.

To confirm the former requirement, simply review the '**_cI**' variable:

```
[ #1 ] $ _cI
#1of2
```

This indicates that the current build has two Cores, and the user is currently communicating with the first.

Another way to check this (and the data channel configuration) is to review the output of the '**list|xbar_cfg**' command:

```
[ #1 ] / list|xbar_cfg

  XBAR LOCK

                |  -  |CORE2|
         LOCK EN|     |  X  |
      LOCK DEPTH|     |  1  |
       UNLOCK EN|     |     |
    UNLOCK DEPTH|     |     |


  XBAR DATA

                |  -  |CORE2|
          TX EN|     |  X  |
   TX FIFO DEPTH|     |  16 |
          RX EN|     |  X  |
   RX FIFO DEPTH|     |  8  |
```

From this output we can see that the data channel between Core #1 and Core #2 is enabled - the forward path from Core #1 to Core #2 has been configured to a FIFO depth of sixteen, and the reverse to a FIFO depth of eight.

6.8.2 Script

First, from Core #1, send some bytes. This doesn't hang, as the forward path FIFO depth is greater than four (i.e. the bytes can all be cached into the FIFO - it doesn't fill - if it did, the function call to 'pipe_tx_byte' that caused it to fill would hang, until a byte was read from it by Core #2).

> The Core # prompt has been included in the following snippets, to better illustrate the example (unfortunately the user won't be able to simply copy-paste these snippets).

```
[ #1 ] $ import("pipe_tx_byte")
[ #1 ] $
[ #1 ] $ pipe_tx_byte(CORE2, 222)
[ #1 ] $ pipe_tx_byte(CORE2, 173)
[ #1 ] $ pipe_tx_byte(CORE2, 190)
[ #1 ] $ pipe_tx_byte(CORE2, 239)
[ #1 ] $
```

Next, switch over to Core #2, and receive!

```
[ #2 ] $ import("pipe_rx_byte")
[ #2 ] $
[ #2 ] $ for i=1,5 do
[ #2 ] $$ print(pipe_rx_byte(CORE1))
[ #2 ] $$ end
222
173
190
239
```

It can be observed that the Core #2 serial terminal has hung, and not returned to the user - this is because only four bytes were sent, but the 'pipe_rx_byte' function was called five times. Simply switch back to Core #1 and send one more byte to 'unblock' the Core #2 serial terminal.

6.8.3 References

API (see page 32)

Command Mode (see page 23)

6.9 Inter-Core Synchronization

The following page provides an example of synchronizing execution between between heterogeneous Cores.

6.9.1 Setup

Similar to the other Inter-Core example, this example requires Lumorphix to be built to support more than one Core, but to have the locking channel (XBar Lock) between at least two of them enabled.

To check this (and the lock channel configuration), review the output of the '**list|xbar_cfg**' command:

```
[ #1 ] / list|xbar_cfg

  XBAR LOCK

                |  -   |CORE2|
         LOCK EN|      |  X  |
      LOCK DEPTH|      |  1  |
       UNLOCK EN|      |     |
    UNLOCK DEPTH|      |     |


  XBAR DATA

                |  -   |CORE2|
          TX EN|      |  X  |
   TX FIFO DEPTH|      |  16 |
          RX EN|      |  X  |
   RX FIFO DEPTH|      |  8  |
```

From this output we can see that the forward locking barrier between Core #1 and Core #2 is enabled - the forward path from Core #1 to Core #2 has been configured to support a single lock (depth of one).

6.9.2 Script

As detailed in the API (see page 32) section, all lock barriers are initialised to the 'unlocked' state upon power-on. Thus, the first example details controlling the execution of Core #2 from Core #1. Again, the core number prompt is included in the example, for clarity.

```
[ #2 ] $ import("all")
[ #2 ] $ unlock(CORE1)
```

At this point, Core #2 will hang, as it attempted to unlock the forward barrier, which was already unlocked. It is essentially now waiting for Core #1 to lock the forward locking barrier (upon which it can then immediately unlock the barrier). Switching over to Core #1:

```
[ #1 ] $ import("all")
[ #1 ] $ lock(CORE2)
```

If the user comms channel is switched back to Core #2, one can confirm Core #2 is no longer hanging and has resumed execution.

Next, the reverse is demonstrated:

```
[ #1 ] $ lock(CORE2)
[ #1 ] $ lock(CORE2)
```

At this point, Core #1 will hang, as it attempted to lock the forward barrier, which it had already locked. It is essentially now waiting for Core #2 to unlock the forward locking barrier (upon which it can then immediately lock the barrier). Switching over to Core #2:

```
[ #2 ] $ unlock(CORE1)
```

Again, switching user comms channel, one can confirm Core #1 is no longer hanging and has resumed execution.

The Lumorphix API offers locking and unlocking functions that do not hang if the attempt fails (non-blocking).

6.9.3 References

API (see page 32)

6.10 Exit Script/Program

The following page provides an example of exiting a script or program.

6.10.1 Example

This is extremely straightforward. Essentially it is the programmatic equivalent of pressing the '**q**' key, except one can output a return value, if required. For example:

```
import("sleep")
for i=1,99999 do print("\tThis is gunna take almost forever to finish !?");
if i > 3 then exit("\n\tNot Forever!"); end; sleep(1) end
    This is gunna take almost forever to finish !?
    This is gunna take almost forever to finish !?
    This is gunna take almost forever to finish !?
    This is gunna take almost forever to finish !?

    Not Forever!
```

Too easy!

6.10.2 References

API (see page 32)

6.11 Non-volatile I/O Config

The following page provides an example of setting the current I/O configuration to persist across reboots.

6.11.1 Setup

So you've reviewed the I/O capabilities (**/ list|io_caps**), and decided which I/O should be configured as which protocol (**/ set|io_type_cfg(...)** or **$ set_io_type_cfg(..., ...)**).

You now want this configuration to persist across reboot cycles. This task is straightforward, simply:

Enter command mode.

```
$ command
```

Save the current I/O configuration as default

```
/ set|start_on_boot_io_cfg
```

Power cycle the device unplug-plug, press the reset button, or run either of the following:

```
/ run|reboot
```

```
/ exit
$ reboot()
```

Enter command mode.

```
$ cmd
```

Confirm the I/O configuration has persisted:

```
/ list|io_type_cfg
```

The same will apply to any related configuration (UART baud, SPI clock frequency, etc).

6.11.2 References

Command Mode (see page 23)

API (see page 32)

6.12 Program Upload

The following page provides an example of uploading and running a program.

6.12.1 Setup

So you've tinkered with the REPL and have flushed out some basic functionality, which you've then finalized by writing it up as a full Lua script in a text file on your local machine. How do you then upload and run this script? Follow these steps:

Enter command mode.

```
$ command
```

Upload the program with the desired name.

```
/ upload|program("my_program.lua")
```

As the uploader utilises the same communications channel as the serial client, you now must close your serial client.

If you are using Tio[21], press:

1. CTRL-t followed by:
2. 'q'

If you are using Putty, simply close the window.

Some serial clients (Teraterm) allow the user to 'Disconnect' without closing the window, which is more convenient for this use case.

If you are using Linux/OSX on your local machine, run the following in your terminal emulator:

```
user@pc> cd <directory of 'program_uploader.py'>
user@pc> python program_uploader.py <your program script> /dev/ttyUSB1
```

21 https://github.com/tio/tio

If you are using Windows on your local machine, run the following in Windows powershell:

```
PS C:> cd <directory of 'program_uploader.py'>
```

(In this example, 'C:\Downloads')

```
PS C:\Downloads> python program_uploader.py
<path to your program script>\<your program script name> COM<port number>
```

See 'GETTING STARTED' section if you are unsure on how to ascertain '<port number>'.

Confirm the program uploader has completed successfully:

```
Upload success!
```

Re-establish serial communications and again, enter command mode.

```
$ command
```

List the saved programs. Your program should be listed (ordered by upload order).

```
/ list|programs

Program 1 : my_program.lua
```

Exit command mode.

```
/ exit
```

Run your program.

```
$ run_program("my_program.lua")
```

```
Hello World!
```

The program uploader script is available here[22]. The Python installer for windows is available here[23].

Ensure that you select 'Customize installation'-'Next' then tick 'Add Python to environment variables' and 'Precompile standard library'.

Ensure that you also install pyserial (after you've install Python, run the following command in Windows Powershell):

```
PS C:> pip install pyserial
```

22 https://brisbanesilicon.com.au/docs/Lumorphix_ProgramUploader.zip
23 https://www.python.org/downloads/windows/

7 Example Programs

The following section includes various Lumorphix example programs.

7.1 gpio_toggle.lua

```
--[[
        Demonstrate GPIO Out toggle (and LED - as it's GPIO1)

        NOTE: stray, beginning of line '--' comments improve
        code readability while still allowing user to copy-paste
        the entire program into the REPL.
--]]


main_program_name = "GPIO OUT Toggle Rate Example"


-- Import required functions from API

quick_import = true
    -- NOTE: change to 'false' to demonstrate
    -- more code import functions, but this will
    -- consume less memory (importing only API
    -- functionality that is used)

if quick_import then
    import("all")
else
    import("reset_all_io_type_cfg")
    import("set_io_type_cfg")
--
    import("set_gpio")
--
    import("sleep")
    import("msleep")
end

-- Program configuration

GPIO_OUT_PIN = 1
LOOP_COUNT = 12


-- Main program begin

print("Program begin: "..main_program_name)


-- Initialise PINS

print("Initialise PINs")
```

```
reset_all_io_type_cfg()
set_io_type_cfg(GPIO_OUT_PIN, GPIO_OUT)

msleep(500)

print("Set GPIO"..GPIO_OUT_PIN.." HIGH")
set_gpio(GPIO_OUT_PIN, HIGH)
sleep(1)

print("Set GPIO"..GPIO_OUT_PIN.." LOW")
set_gpio(GPIO_OUT_PIN, LOW)
sleep(1)

-- Perform output toggle loop

for i=1,LOOP_COUNT do
    print("Toggle GPIO"..GPIO_OUT_PIN)
    set_gpio(GPIO_OUT_PIN, TOGGLE)
    msleep(500)
end

sleep(1)

print("Program end: "..main_program_name)
```

7.2 gpio_interrupt.lua

```
--[[
        Demonstrate GPIO interrupts

        NOTE: stray, beginning of line '--' comments improve
        code readability while still allowing user to copy-paste
        the entire program into the REPL.
--]]


main_program_name = "GPIO Interrupt Example"


-- Import required functions from API

quick_import = false
    -- NOTE: change to 'true' to demonstrate
    -- faster & less code import, but this will
    -- consume more memory (importing API
    -- functionality that is unused)

if quick_import then
    import("all")
else
    import("global_interrupt_enable")
    import("global_interrupt_disable")
    import("enable_interrupt_types_for_pin")
    import("set_interrupt_types_for_pin")
    import("get_interrupts_on_pin")
    import("ack_interrupt_types_on_pin")
--
    import("reset_all_io_type_cfg")
    import("set_io_type_cfg")
--
    import("set_gpio")
    import("get_gpio")
--
    import("sleep")
    import("msleep")
end


-- Helper functions

function board_setup_msg (prefix)
    msg = prefix.." '"..main_program_name.."' requires feedback resistor"
    msg = msg .."(< 100 Ohm) between PIN"..GPIO_OUT_PIN.." and PIN"
```

```
    return msg..GPIO_IN_PIN.."."
end


-- Program configuration

GPIO_OUT_PIN = PIN2
GPIO_IN_PIN = PIN1
GPIO_IN_PIN_INTERRUPT_BITMASK = PIN1_BITMASK

LOOP_COUNT = 60
LOOP_SLEEP_MS = 500

PERFORMANCE_TEST_DURATION_MILLISEC = 1000


-- Global variables

interrupt_occurred_base_msg = "\tI] PIN"..GPIO_IN_PIN.."! Type(s): "
set_gpio_high_msg = "M] GPIO"..GPIO_OUT_PIN.." HIGH"
set_gpio_low_msg = "M] GPIO"..GPIO_OUT_PIN.." LOW"

doing_performance_test = false
intrpt_cntr = 0


-- Main program begin

print("Program begin: "..main_program_name)


print(board_setup_msg("Important!"))
sleep(1)


-- Initialise REPL mode


-- Initialise Board State

print("Initialise Board State")

global_interrupt_disable()
reset_all_io_type_cfg()
sleep(1)


-- Initialise PINS

print("Initialise I/Os")
set_io_type_cfg(GPIO_OUT_PIN, GPIO_OUT)
set_io_type_cfg(GPIO_IN_PIN, GPIO_IN)
```

```
sleep(1)


-- Confirm external board setup

print("Confirm external board setup")

set_gpio(GPIO_OUT_PIN, HIGH)
gpio_level = get_gpio(GPIO_IN_PIN)
if gpio_level ~= HIGH then
    exit(board_setup_msg("Board setup error!"))
end

set_gpio(GPIO_OUT_PIN, LOW)
gpio_level = get_gpio(GPIO_IN_PIN)
if gpio_level ~= LOW then
    exit(board_setup_msg("Board setup error!"))
end


-- Initialise Interrupts

print("Initialise Interrupts")

set_interrupt_types_for_pin(GPIO_IN_PIN, NONE)
ack_interrupt_types_on_pin(GPIO_IN_PIN, 0xFF)
    -- NOTE: acknowledge any existing interrupts
sleep(1)


-- Define Interrupt Handler

print("Define Interrupt Handler")

function interrupt_handler(interrupt_vector)
    intrpt_cntr = intrpt_cntr + 1
--
    if not doing_performance_test then
        if (interrupt_vector & GPIO_IN_PIN_INTERRUPT_BITMASK) ~= 0 then
            gpio_interrupt_vector = get_interrupts_on_pin(GPIO_IN_PIN)
            ack_interrupt_types_on_pin(GPIO_IN_PIN, gpio_interrupt_vector)
                -- NOTE: ack all interrupts that occurred
--
            -- NOTE: print GPIO interrupts that occurred
            interrupt_types_str = ""
            if (gpio_interrupt_vector & GPIO_INTRPT_RISING_EDGE) ~= 0 then
                interrupt_types_str = "R | "
            end
            if (gpio_interrupt_vector & GPIO_INTRPT_FALLING_EDGE) ~= 0 then
                interrupt_types_str = interrupt_types_str.."F | "
            end
            if (gpio_interrupt_vector & GPIO_INTRPT_LOW) ~= 0 then
```

```
                interrupt_types_str = interrupt_types_str.."L | "
            end
            if (gpio_interrupt_vector & GPIO_INTRPT_HIGH) ~= 0 then
                interrupt_types_str = interrupt_types_str.."H | "
            end
--
            print(interrupt_occurred_base_msg..interrupt_types_str..intrpt_cntr)
        else
            -- NOTE: shouldn't get here!
--
            print("\tI] Interrupted on other PIN than PIN"..GPIO_IN_PIN.." ?!")
            print("\tI] interrupt_vector:"..interrupt_vector)
        end
    end
end


-- Bind Interrupt Handler

print("Bind Interrupt Handler")

interrupt.bind(interrupt_handler)


-- Perform toggle / interrupt loop

for i=1,LOOP_COUNT do
    print("M] <sleep "..LOOP_SLEEP_MS.." ms>")
    msleep(LOOP_SLEEP_MS)
--
    if (i % 6) == 0 then
        print(set_gpio_high_msg)
        set_gpio(GPIO_OUT_PIN, HIGH)
    elseif (i % 3) == 0 then
        print(set_gpio_low_msg)
        set_gpio(GPIO_OUT_PIN, LOW)
    end
--
    if i == 10 then
        print("M] Global interrupt enable")
        global_interrupt_enable()
    elseif i == 20 then
        print("M] Enable only falling edge interrupt on GPIO"..GPIO_IN_PIN)
        set_interrupt_types_for_pin(GPIO_IN_PIN, GPIO_INTRPT_FALLING_EDGE)
    elseif i == 35 then
        print("M] Enable also rising edge interrupt on GPIO"..GPIO_IN_PIN)
        enable_interrupt_types_for_pin(GPIO_IN_PIN, GPIO_INTRPT_RISING_EDGE)
    elseif i == 50 then
        print("M] Enable only VCC and GND interrupt on GPIO"..GPIO_IN_PIN)
        set_interrupt_types_for_pin(GPIO_IN_PIN, GPIO_INTRPT_HIGH | GPIO_INTRPT_LOW)
    elseif i == LOOP_COUNT then
        set_interrupt_types_for_pin(GPIO_IN_PIN, NONE)
```

```
        sleep(1)
--
        print("M] Perform performance test")
        set_gpio(GPIO_OUT_PIN, LOW)
        intrpt_cntr = 0
        doing_performance_test = true
        set_interrupt_types_for_pin(GPIO_IN_PIN, GPIO_INTRPT_LOW)
        msleep(PERFORMANCE_TEST_DURATION_MILLISEC)
    end
end


-- End Program

global_interrupt_disable()

perf_res_msg = (intrpt_cntr / PERFORMANCE_TEST_DURATION_MILLISEC).." kHz"
print("Peformance results: "..perf_res_msg)
print("Program end: "..main_program_name)
```

7.3 led_fade_inout.lua

```
--[[
        Demonstrate LED (GPIO1) fade in / out

        NOTE: stray, beginning of line '--' comments improve
        code readability while still allowing user to copy-paste
        the entire program into the REPL.
--]]


main_program_name = "LED Fade"


-- Import required functions from API

import("reset_all_io_type_cfg")
import("set_io_type_cfg")
import("set_pwm")
import("set_gpio")
import("sleep")
import("msleep_f")


-- Program configuration

PWM_PIN = 1
LOOP_COUNT = 10


-- Main program begin

print("Program begin: "..main_program_name)


-- Initialise PINS

print("Initialise PINs")

reset_all_io_type_cfg()
set_io_type_cfg(PWM_PIN, PWM)
set_io_type_cfg(PIN2, GPIO_OUT)
set_io_type_cfg(PIN3, GPIO_OUT)

sleep(1)


-- Helper functions
```

```
function is_even(x)
    return ((x & 1) == 0)
end


-- Set other LEDs to GND

set_gpio(PIN2, LOW)
set_gpio(PIN3, LOW)


-- Perform LED fade in/out

print("Begin Fade in/out")

for i=1,LOOP_COUNT do
    for j=5,PWM_MAX do
        if is_even(i) then
            set_pwm(PWM_PIN, j)
            msleep_f(100/j)
        else
            set_pwm(PWM_PIN, (PWM_MAX-j))
            msleep_f(100/((PWM_MAX+5)-j))
        end
    end
end


-- Cleanup (free memory)

main_program_name = nil

PWM_PIN = nil
LOOP_COUNT = nil

reset_all_io_type_cfg = nil
set_io_type_cfg = nil
set_pwm = nil
sleep = nil
msleep_f = nil

collectgarbage()

print("Program end")
```

7.4 fibonacci_nonrecursive.lua

```
--[[
        Demonstrate Fibonacci Sequence calculated without recursion (fast in Lua)

        NOTE: stray, beginning of line '--' comments improve
        code readability while still allowing user to copy-paste
        the entire program into the REPL.
--]]


main_program_name = "Fibonacci Sequence Non-Recursive Example"


-- Program configuration

n = 30


-- Helper functions

function FiboNonRecursive(n)
    a = 0
    b = 1
    c = (n >> 1)
    for i=1,c do
        a=a+b
        b=a+b
    end
--
    if (n & 1) == 1 then
        return b;
    else
        return a;
    end
end



-- Main program begin

print("Program begin: "..main_program_name)


-- Calculate Fibonacci

fibo_n = FiboNonRecursive(n)
```

```
print("Fibo("..n..")": "..fibo_n)


-- Ends

print("Program end: "..main_program_name)
```

7.5 fibonacci_recursive.lua

```
--[[
        Demonstrate Fibonacci Sequence calculated via recursion (very slow in Lua)

        NOTE: stray, beginning of line '--' comments improve
        code readability while still allowing user to copy-paste
        the entire program into the REPL.
--]]


main_program_name = "Fibonacci Sequence Recursive (slow) Example"


-- Program configuration

n = 10


-- Helper functions

function FiboRecursive(n)
    if (n == 1) then
        return 1;
    end
    if (n == 0) then
        return 0;
    end
--
    return FiboRecursive(n-1) + FiboRecursive(n-2);
end


-- Main program begin

print("Program begin: "..main_program_name)


-- Calculate Fibonacci

fibo_n = FiboRecursive(n)

print("Fibo("..n.."): "..fibo_n)


-- Ends

print("Program end: "..main_program_name)
```

7.6 sseg_regs.lua

```
--[[
        SSEG register definiton for Duinotech display.
        See MAX7219/MAX7221 "Serially Interfaced, 8-Digit
        LED Display Drivers" document.

        https://www.analog.com/media/en/technical-documentation/
        data-sheets/MAX7219-MAX7221.pdf.

        NOTE: variables are declared as 'local' to ensure they
        do not consume memory after the program exits...
--]]

-- SSEG registers

local digit_regs = {}
digit_regs[1] = 0x01
digit_regs[2] = 0x02
digit_regs[3] = 0x03
digit_regs[4] = 0x04
digit_regs[5] = 0x05
digit_regs[6] = 0x06
digit_regs[7] = 0x07
digit_regs[8] = 0x08

local mode_regs = {}
mode_regs["REG_DECODE_MODE_CTRL"]       = 0x09
mode_regs["REG_INTENSITY_CTRL"]         = 0x0A
mode_regs["REG_SCAN_LIMIT_CTRL"]        = 0x0B
mode_regs["REG_OPERATION_MODE_CTRL"]    = 0x0C
mode_regs["REG_DISPLAY_TEST_MODE_CTRL"] = 0x0F

return digit_regs, mode_regs
```

7.7 spi_out.lua

```
--[[
        Demonstrate SPI output by counting on a
        SSEG display (Duinotech display).

        NOTE: stray, beginning of line '--' comments improve
        code readability while still allowing user to copy-paste
        the entire program into the REPL.

        NOTE: Depending upon 'sseg_regs_uploaded', this program
        requires the program 'sseg_regs.lua' to be present on Lumorphix.
--]]


main_program_name = "SPI OUT Example"

sseg_regs_uploaded = false
    -- NOTE: change to 'true' if 'sseg_regs.lua' has been
    -- uploaded to Lumorphix


-- Dependencies

import("reset_all_io_type_cfg")
import("set_io_type_cfg")
import("spi_tx_int")
import("msleep")

import("string", "format")

if sseg_regs_uploaded then
    digit_regs, mode_regs = run_program("sseg_regs.lua")
else
    digit_regs = {}
    digit_regs[1] = 0x01
    digit_regs[2] = 0x02
    digit_regs[3] = 0x03
    digit_regs[4] = 0x04
    digit_regs[5] = 0x05
    digit_regs[6] = 0x06
    digit_regs[7] = 0x07
    digit_regs[8] = 0x08

    mode_regs = {}
    mode_regs["REG_DECODE_MODE_CTRL"]       = 0x09
    mode_regs["REG_INTENSITY_CTRL"]         = 0x0A
    mode_regs["REG_SCAN_LIMIT_CTRL"]        = 0x0B
```

```
    mode_regs["REG_OPERATION_MODE_CTRL"]     = 0x0C
    mode_regs["REG_DISPLAY_TEST_MODE_CTRL"] = 0x0F
end



-- SSEG register values

DISPLAY_TEST_MODE_OFF           = 0x00
DISPLAY_NORMAL_OPERATION_MODE   = 0x01
DISPLAY_DECODE_MODE_CODE_B      = 0xFF
DISPLAY_SCAN_ALL_DIGITS         = 0x07
DISPLAY_INTENSITY_LOW           = 0x03


-- Program configuration

SPI_OUT_PIN = 1

LOOP_COUNT = 100
LOOP_DURATION_MS = 50


-- Helper functions

function spi_cmnd (register, value)
    return (register << 8) | value
end


-- Main program begin

print("Program begin: "..main_program_name)


-- Initialise PINS

print("Initialise PINs")

reset_all_io_type_cfg()
set_io_type_cfg(SPI_OUT_PIN, SPI_OUT)

msleep(500)


-- Initialise SSEG

print("Initialise SSEG")

cmnds = {}
cmnds[1] = spi_cmnd(mode_regs["REG_DISPLAY_TEST_MODE_CTRL"], DISPLAY_TEST_MODE_OFF)
```

```
cmnds[2] = spi_cmnd(mode_regs["REG_OPERATION_MODE_CTRL"],
DISPLAY_NORMAL_OPERATION_MODE)
cmnds[3] = spi_cmnd(mode_regs["REG_DECODE_MODE_CTRL"], DISPLAY_DECODE_MODE_CODE_B)
cmnds[4] = spi_cmnd(mode_regs["REG_SCAN_LIMIT_CTRL"], DISPLAY_SCAN_ALL_DIGITS)
cmnds[5] = spi_cmnd(mode_regs["REG_INTENSITY_CTRL"], DISPLAY_INTENSITY_LOW)

for i=1,#cmnds do
    print("SPI CMD "..i..": 0x"..string.format("%.4x", cmnds[i]))
--
    spi_tx_int(SPI_OUT_PIN, cmnds[i])
    msleep(50)
end

msleep(500)


-- Perform counting loop

print("Perform counting loop")

target_digit = 1
target_digit_value = 0
for i = 1, LOOP_COUNT do
    spi_tx_int(SPI_OUT_PIN, spi_cmnd(digit_regs[target_digit], target_digit_value))
--
    msleep(LOOP_DURATION_MS);
--
    target_digit = target_digit + 1
    if target_digit > #digit_regs then
        target_digit = 1
    end
--
    target_digit_value = target_digit_value + 1
    if target_digit_value > 9 then
        target_digit_value = 0
    end
end


print("Program end: "..main_program_name)
```

7.8 uart_rx_interrupt.lua

```
--[[
        Demonstrate UART Rx and UART Rx interrupts
--]]


main_program_name = "UART Rx and UART Rx Interrupt Example"


-- Import required functions from API

quick_import = false
    -- NOTE: change to 'true' to demonstrate
    -- faster & less code import, but this will
    -- consume more memory (importing API
    -- functionality that is unused)

if quick_import then
    import("all")
else
    import("global_interrupt_enable")
    import("global_interrupt_disable")
    import("enable_interrupt_types_for_pin")
    import("set_interrupt_types_for_pin")
    import("get_interrupts_on_pin")
    import("ack_interrupt_types_on_pin")
--
    import("reset_all_io_type_cfg")
    import("set_io_type_cfg")
--
    import("uart_rx_char")
--
    import("sleep")
    import("msleep")
end

-- Program configuration

UART_RX_PIN = PIN2
UART_RX_PIN_INTERRUPT_BITMASK = PIN2_BITMASK

LOOP_COUNT = 1000
LOOP_SLEEP_MS = 20


-- Main program begin
```

```
print("Program begin: "..main_program_name)
sleep(1)


-- Initialise Board State

print("Initialise Board State")

global_interrupt_disable()
reset_all_io_type_cfg()
sleep(1)


-- Initialise PINS

print("Initialise I/Os")
set_io_type_cfg(UART_RX_PIN, UART_IN)
sleep(1)


-- Initialise Interrupts

print("Initialise Interrupts")

set_interrupt_types_for_pin(UART_RX_PIN, NONE)
ack_interrupt_types_on_pin(UART_RX_PIN, 0xFF)
    -- NOTE: acknowledge any existing interrupts
sleep(1)


-- Define Interrupts Handler

print("Define Interrupt Handler")

function interrupt_handler(interrupt_vector)
    if (interrupt_vector & UART_RX_PIN_INTERRUPT_BITMASK) ~= 0 then
            -- NOTE: no need to get interrupt vector, as
            -- there is only one for UART RX (UART_RX_INTRPT_DATA_AVAILABLE)
            -- NOTE: no need to ACK, as interrupt will clear
            -- automatically when the RX dat is read
--
        int_vec = get_interrupts_on_pin(UART_RX_PIN)
        if (int_vec & UART_RX_INTRPT_DATA_AVAILABLE) == 0 then
            -- NOTE: don't really need to do this...
            msg = "\t[Interrupt] PIN"..UART_RX_PIN.." interrupt vector "
            print(msg.."doesn't contain 'UART_RX_INTRPT_DATA_AVAILABLE' ?!")
        end
--
        print("\t[Interrupt] UART RX: '"..uart_rx_char(UART_RX_PIN).."'")
    else
        -- NOTE: shouldn't get here!
--
```

```
        print("\t[Interrupt] Interrupted on other PIN than PIN"..UART_RX_PIN.." ?!")
        print("\t[Interrupt] interrupt_vector:"..interrupt_vector)
    end
end
interrupt.bind(interrupt_handler)


-- Perform output toggle loop

for i=1,LOOP_COUNT do
    if (i % 100) == 0 then
        print("[Main Loop] <tick>")
    end
    if i == 150 then
        print("[Main Loop] Global interrupt enable")
        global_interrupt_enable()
    end
    if i == 250 then
        print("[Main Loop] Enable RX data received interrupt on UART"..UART_RX_PIN)
        set_interrupt_types_for_pin(UART_RX_PIN, UART_RX_INTRPT_DATA_AVAILABLE)
    end
--
    msleep(LOOP_SLEEP_MS)
end


-- End Program

global_interrupt_disable()

print("Program end: "..main_program_name)
```

7.9 exit_prog.lua

```
--[[
        Demonstrate exiting a running program or script (if copy-
        pasted into REPL).

        NOTE: stray, beginning of line '--' comments improve
        code readability while still allowing user to copy-paste
        the entire program into the REPL.
--]]


main_program_name = "Exit Program Example"

-- Import required functions from API

import("msleep")


-- Program configuration

PRIMARY_LOOP_COUNT = 1000000
SECONDARY_LOOP_COUNT = 5


-- Main program begin

print("Program begin: "..main_program_name)


-- Initialise PINS


-- Perform output toggle loop

for i=1,PRIMARY_LOOP_COUNT do
    for j=1,SECONDARY_LOOP_COUNT do
        print("Loop! I: "..i.. ", J: "..j)
--
        msleep(100)
        if i > 7 then
            if j > 3 then
                exit("Bye for now!")
            end
        end
    end
end
```

```
-- NOTE: won't get here!

print("Program end: "..main_program_name)
```

7.10 exit_prog_nested.lua

```
--[[
        Demonstrate nested program exit.

        NOTE: This program requires the script
        'exit_prog.lua' to be present on Lumorphix as
        a program.
--]]


main_program_name = "Exit Program Nested Example"


-- Dependencies

unused_unassigned = run_program("exit_prog.lua")


-- NOTE: we shouldn't end up here!

import("sleep")

for i=1,10 do
        print("How did I get here?!")
        sleep(2)
end

print("Program end: "..main_program_name)
```

7.11 prog_err.lua

```
--[[
        Demonstrate a programming error.

        NOTE: stray, beginning of line '--' comments improve
        code readability while still allowing user to copy-paste
        the entire program into the REPL.
--]]

if true then
--
    local a_string = "blahblahblah"
    local oops_not_allowed = a_string..false
        -- NOTE: not allowed!!
--
end
```

8 Beta Release Info

The following section details the Beta releases of Lumorphix.

8.1 Releases

The following table provides a summary of the supported hardware, software requirements, and release schedule of Lumorphix Beta releases.

Release ID	Release Date	Silicon	Supported Hardware	Software Requirements

02.001	24/02/2025	GOWIN	SiPeed Tang Nano 9k[24]	• Python[25]
				• FirmwareUploader[26]
				• ProgramUploader[27] (if user requires program functionality).
				• GOWIN Programmer[28] (Windows only)
				  • Additional instructions are available here[29].
				• OpenFPGALoader[30] (Linux only)
				  • Additional instructions are available here[31] and here[32].

01.001	24/01/2025	AMD / Xilinx	Digilent Arty S7[33] [i]	• AMD/Xilinx Vivado[34] [ii]

All Lumorphix Beta releases require a serial terminal emulator program. See Serial Terminal Setup (see page 17) for more information.

24 https://wiki.sipeed.com/hardware/en/tang/Tang-Nano-9K/Nano-9K.html
25 https://www.python.org/
26 https://brisbanesilicon.com.au/utilities/firmware-uploader.py
27 https://brisbanesilicon.com.au/utilities/program-uploader.py
28 https://dl.sipeed.com/shareURL/TANG/programmer
29 https://wiki.sipeed.com/hardware/en/tang/common-doc/install-the-ide.html
30 https://github.com/trabucayre/openFPGALoader
31 https://trabucayre.github.io/openFPGALoader/guide/install.html
32 https://wiki.sipeed.com/hardware/en/tang/common-doc/flash-in-linux.html
33 https://digilent.com/reference/programmable-logic/arty-s7/start
34 https://www.xilinx.com/support/download.html

8.1.1 References

[i] Supported ARTY FPGA variants are as follows:

S7-25, S7-50

[ii] While the Enterprise edition of Vivado is supported, at a minimum all that is required is Vivado Lab Solutions.

8.2 Feature List

Release ID	Feature Info
02.001	• Non-volatile storage supported
01.001	• Non-volatile storage unsupported

8.3 Hardware Targets

The following section details any hardware-specific information for the supported Beta release target hardware.

8.3.1 Hardware Specific Information

Select the appropriate link within the 'Hardware Specific Information' column to review any information specific to your target hardware.

Target Hardware	Hardware Specific Information
Tang Nano 9k	Tang Nano 9k (see page 101)
Digilent ARTY-S7	ARTY-S7 (see page 104)

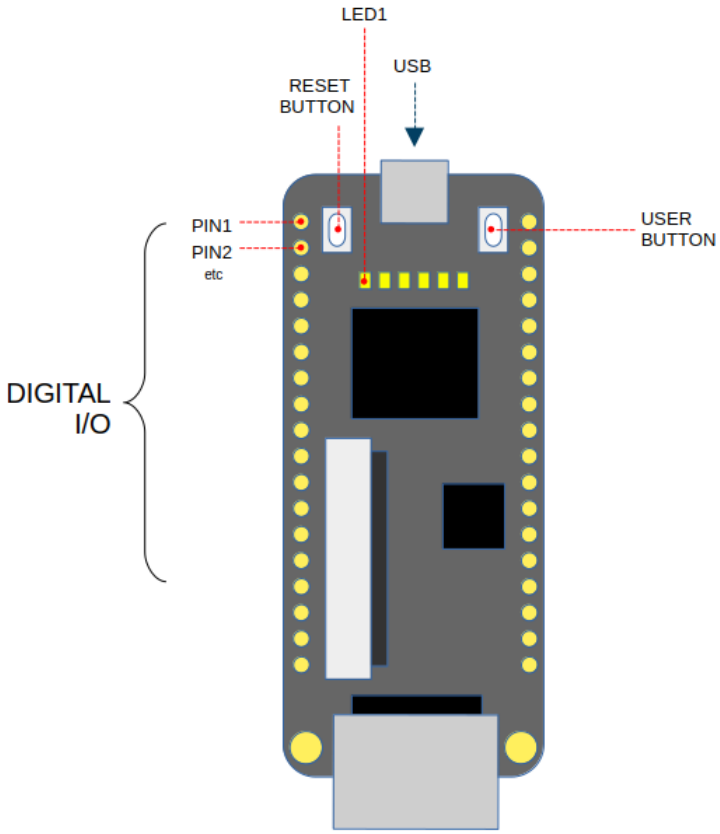
8.3.2 Tang Nano 9k

The following page details hardware-specific information for the Tang Nano 9k Beta Release target hardware.

8.3.2.1 Setup

There is no custom setup for the Tang Nano 9k board.

8.3.2.2 Board Layout (generic)



8.3.2.2.1 Description

8.3.2.2.1.1 Reset button

This button performs a soft-reset of the Core with which the user is currently interacting.

8.3.2.2.1.2 User button

The functionality of this button is dependent upon both the Lumorphix boot state, and the loaded variant of Lumorphix Beta Release - specifically the number of Cores it contains. Irrespective of the number of Cores, the User button can be used to perform the following actions during boot / power-on:

User Button Actions	Functionality
1. Hold button prior to power-on. 2. Plug the Tang Nano into your PC via the USB. 3. Release button within 3 seconds.	Places the Tang Nano into 'Firmware Update' mode.
1. Hold prior to power-on. 2. Plug the Tang Nano into your PC via the USB. 3. Keep the User Button held until boot completes.	Resets the Lumorphix Flash region (i.e. uploaded programs, boot I/O state etc).

Once Lumorphix has booted, the functionality of the User Button can be summarized as such (Beta Release / Number of Cores dependent):

Cores	User Button Functionality
1	When configured as an input (i.e. GPIO IN), PIN1 will have the value of Digital I/O1 **OR'd** with the User Button.
2	Pressing the User Button will cycle the User Comms to the other Core.

8.3.2.2.1.3 LEDs

LED	Description
1	Heartbeat
2	CORE1 I/O Activity indicator
3	CORE1 Memory Access indicator
4	CORE1, PIN3 state

LED	Description
5	CORE1, PIN2 state
6	CORE1, PIN1 state

8.3.2.3 Hard Reset

To perform a hard reset of this hardware, unplug-plug the board.

8.3.3 ARTY-S7

The following page details hardware-specific information for the ARTY-S7 Beta release target hardware.

8.3.3.1 Setup

The custom setup for the ARTY-S7 board is as follows:

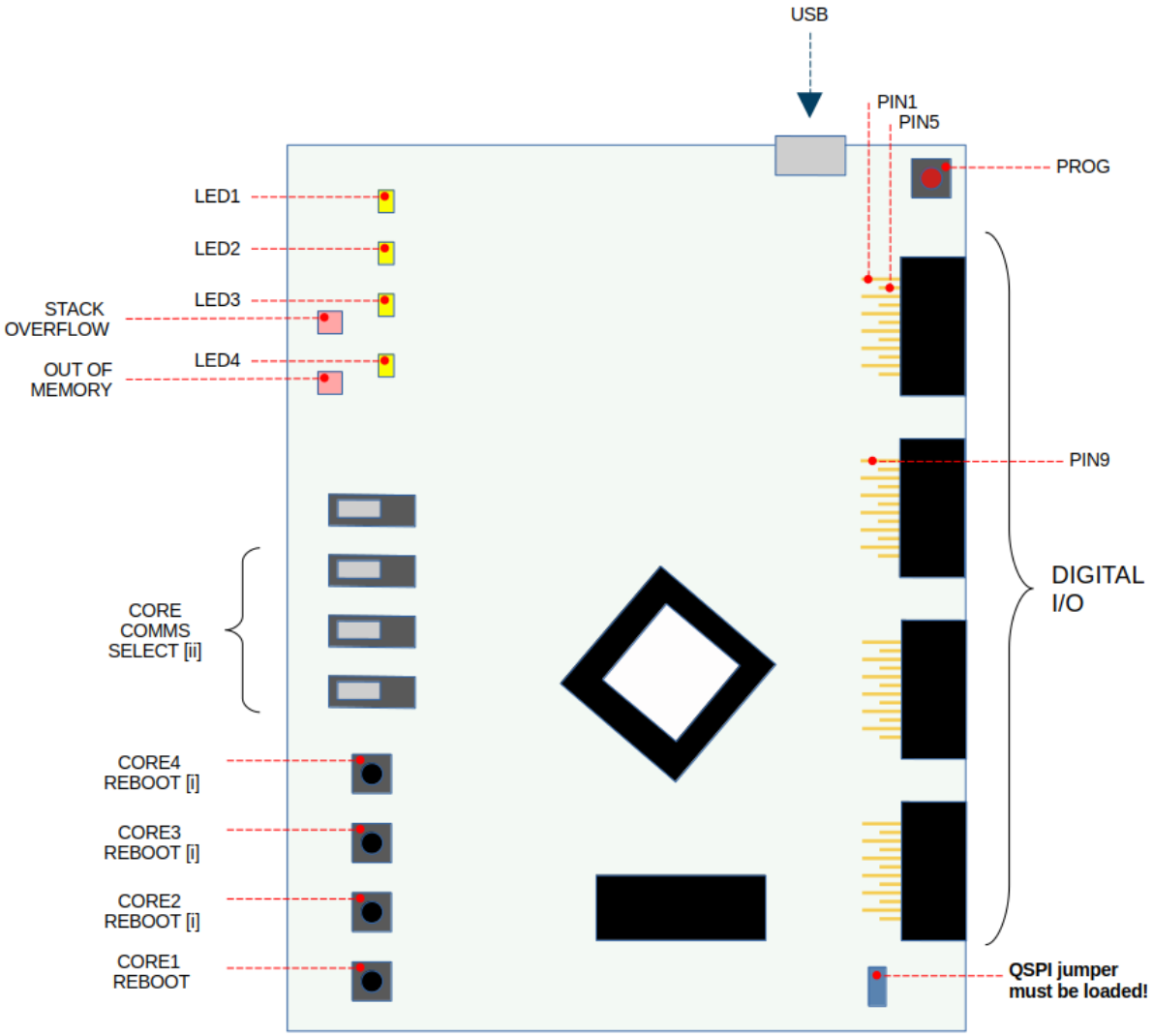
1. Ensure the '**QSPI**' jumper is loaded, as the FPGA bitstream will be loaded (and Lumorphix will boot) from flash.

8.3.3.2 Configuration Memory Part

The following table lists the Configuration Memory Part ID for each variant of the 'ARTY-S7' board.

Board Variant	Configuration Memory Part
s25	s25fl128sxxxxxx0-spi-x1_x2_x4
s50	s25fl128sxxxxxx0-spi-x1_x2_x4

8.3.3.3 Board Layout (generic)



3 ARTY-S7 Board Layout

8.3.3.3.1 Description

8.3.3.3.1.1 PROG button

This button performs a hard-reset of the board (re-programs the FPGA - press twice).

8.3.3.3.1.2 Reboot buttons

These buttons to reboot the corresponding CORE (maximum of 4).

8.3.3.3.1.3 Core Comms select

These switches control which CORE is currently communicating with the users PC (via the terminal program). Together, they effectively form a binary number (lowest switch in the image above, ‘Switch 1’, is ‘Bit 1’), corresponding to the CORE number, as such:

Switch 3 [ii]	Switch 2 [ii]	Switch 1 [ii]	USER COMMUNICATING WITH CORE
OFF	OFF	OFF	1
OFF	OFF	ON	2
OFF	ON	OFF	3
OFF	ON	ON	4
ON	OFF	OFF	5
ON	OFF	ON	6
ON	ON	OFF	7
ON	ON	ON	8

Cores	Switch 1 Used	Switch 2 Used	Switch 3 Used
1	No	No	No
2	Yes	No	No
4	Yes	Yes	No
8	Yes	Yes	Yes

8.3.3.3.1.4 LEDs

LED	Description
1	Heartbeat
2	CORE1 I/O Activity indicator
3	CORE1 Memory Access indicator
4	CORE1, PIN1 state
Stack Overflow	Remains RED if any CORE has failed with stack overflow. Cleared once the offending CORE(s) are reset.
Out of Memory	Remains RED if any CORE has failed with out of memory error. Cleared once the offending CORE(s) are reset.

8.3.3.3.2 References

[i] It is possible that not all of the REBOOT buttons are utilized (minimum one, maximum four). This is dependent upon the Lumorphix Beta release firmware variant that was loaded onto the ARTY-S7 (and therefore the configured number of CORES).

[ii] It is possible that some or all of the CORE COMMS SELECT switches are unused (maximum three). This is again dependent upon the Lumorphix Beta release firmware variant that was loaded onto the ARTY-S7 (and therefore the configured number of CORES).

8.3.3.4 Hard Reset

To perform a hard reset of this hardware, press the ‘PROG’ button twice.

9 Other

The following section details any other information related to the Lumorphix product, which hasn’t already been covered in the previous documentation sections.

9.1 Lua

This page provides a summary of the scripting language supported by Lumorphix, Lua[35].

Lua is an extremely simple, yet powerfully expressive scripting language. We recommend you read the documentation[36] to get started with Lua. Or simply google 'Lua primer', or something similar - there are tons of Lua tutorials freely available online!

9.1.1 Logo

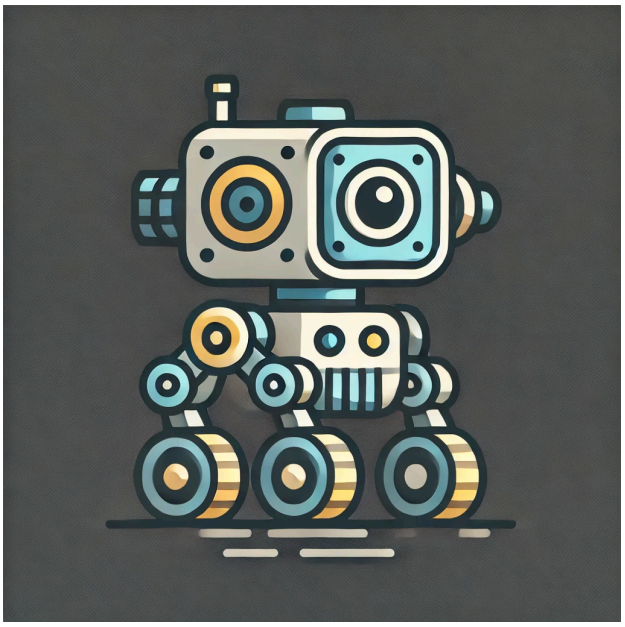
9.1.2 Licence

35 https://www.lua.org/about.html
36 https://www.lua.org/docs.html
37 http://Lua.org

9.2 Company Logo

4 BrisbaneSilicon Logo

9.3 Product Logo



5 Lumorphix Logo

9.4 Copyright

9.4.1 Credits

Firmware and Software (excl. Lua interpreter) by Craig Haywood.

Hardware by William Schofield.

9.4.2 Copyright