

ELM11 Datasheet

A microcontroller board programmed in Lua

Embdedded **L**ua **M**achine



BRisbaneSilicon

www.brisbanesilicon.com.au

Table of Contents

1	About	3
1.1	Overview	3
1.2	Image	4
1.3	Specifications	5
2	Getting Started	8
2.1	Hardware	9
2.2	PC	10
2.3	Connect!	12
2.4	Serial Terminal Setup	16
3	Boot Information	18
3.1	Boot Log Description	18
4	REPL Mode	20
4.1	Prompt	20
4.2	Example	20
4.3	User Interrupt	21
4.4	History	21
4.5	Summary	22
5	Command Mode	23
5.1	Activation	23
5.2	Prompt	23
5.3	Usage	23
6	API	32
6.1	Constants	33
6.2	Functions	37
6.3	Variables	50
7	Example Usage	51
8	Hardware Overlays	52
8.1	O1 (ELM11 shipped default)	53
8.2	O2	55

1 About

1.1 Overview

The ELM11 is a microcontroller board that is programmed in Lua. This document details the following:

1. Specifications of the board.
2. Getting started with the board.
3. Reviewing the boot information.
4. Interacting with the Lua REPL.
5. Overview of 'Command Mode'.
6. Description of the Lua API.
7. Example usage of the Lua API within Lua programs.
8. Available Hardware Overlays.

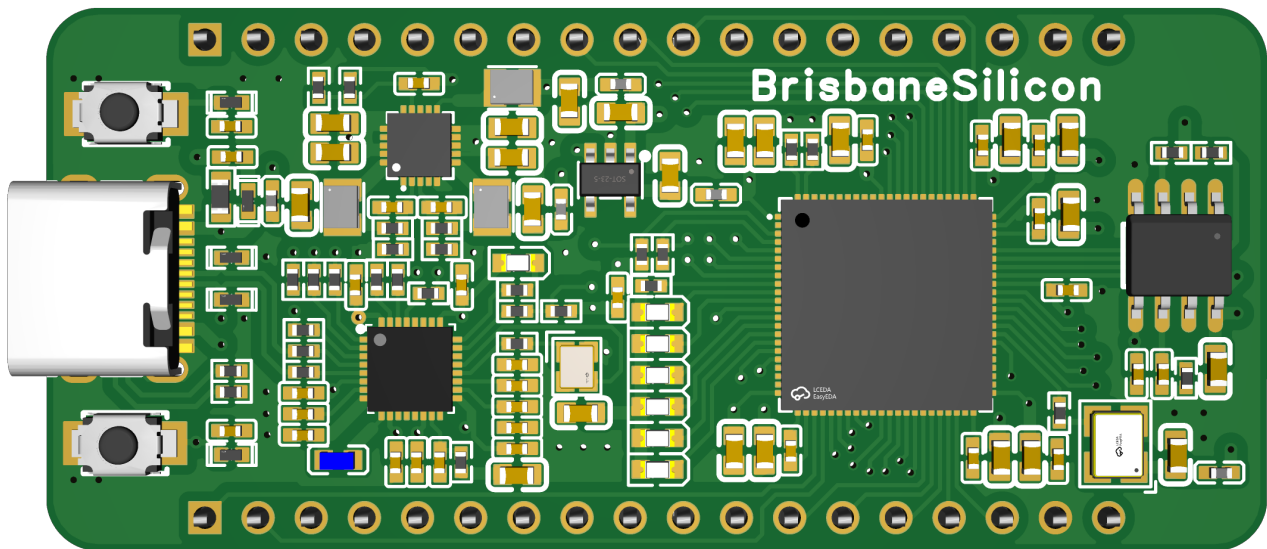


An example of the usage of 'Command Mode' includes the following:

- Query hardware capabilities.
- Configure hardware, e.g. I/O type.
- Upload a Lua program to flash.
- Reboot the board.

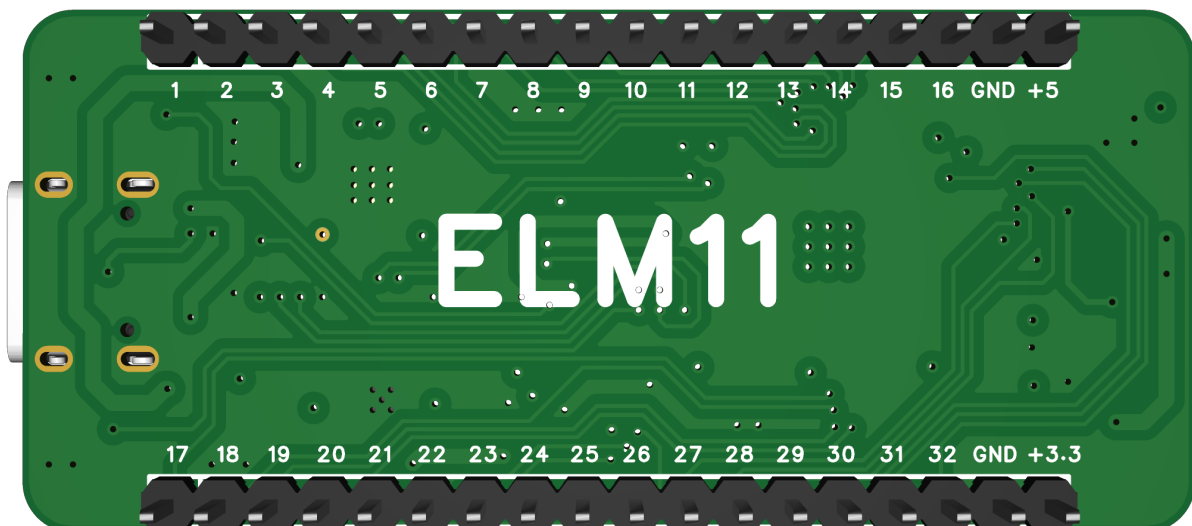
1.2 Image

1.2.1 Top



1 Top - ELM11

1.2.2 Bottom



2 Bottom - ELM11

1.3 Specifications

The specifications of the ELM11 are as follows:

- Executes Lua, from either:
 - REPL
 - *There a standalone REPL for each CPU core.*
 - Program(s) stored on the Flash
- CPU customization
 - Frequency
 - Number of cores
 - Hardware-acceleration of Lua VM
- Support for standard digital I/O protocols, including:
 - GPIO, PWM, UART, SPI, I2C
- Interrupts, including:
 - GPIO level and transition
 - UART, SPI and I2C receive data
- Timers
- Watchdog
 - Hardware-based
- Run-time reconfigurable hardware
 - For example, I/O type, frequency, boot configuration etc. See section [Command Mode](#) (see page 23) for more information.
- Compile-time reconfigurable hardware
 - For example, CPU frequency, I/O run-time capabilities etc.
 - There are multiple 'hardware overlays' available with predefined hardware capabilities. See the following page for an example (that which ships with the ELM11) or section [Hardware Overlays](#) (see page 52) for further information.

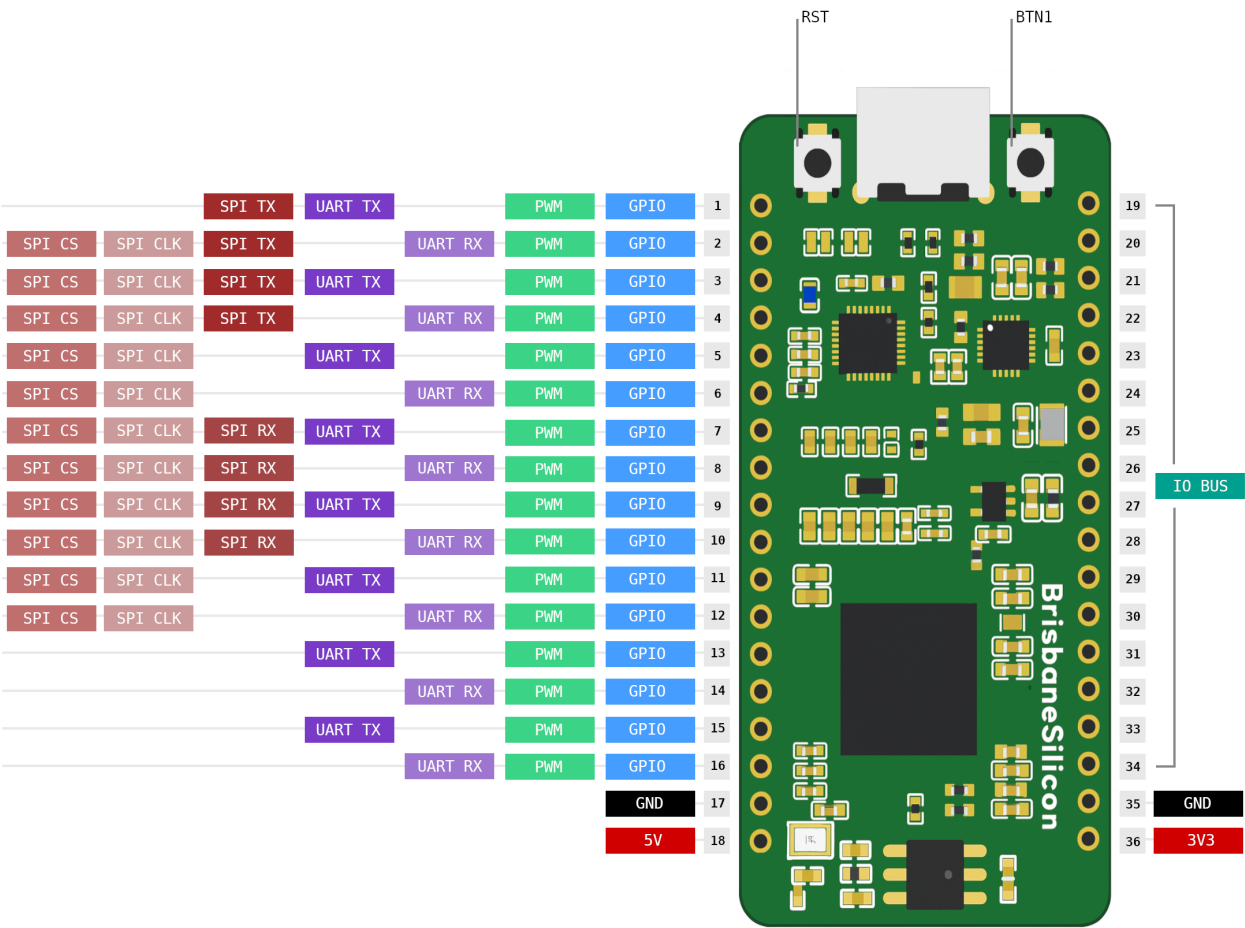
1.3.1 Hardware Overlay Example

An example of a hardware overlay is as follows:

1.3.1.1 CPU

Cores	1
Frequency	66 MHz
Heap	1 MB
Stack	40 KB
PINs	16
IO Bus	Write-only
VM Acceleration	No
Software Interrupts	All PINs
Hardware I/O Buffers	All PINs Skid-Buffer
UART Width	All PINs 8-bit
SPI Width	All PINs 8-bit
Hardware Watchdog	Yes
Performance Timer	Yes
General Timer	Yes

1.3.1.2 Pinout



3 Pinout of Hardware Overlay O1 (ELM11 shipped default)

2 Getting Started

The following section details the initial setup of the ELM11. Begin with the [Hardware \(see page 9\)](#) section.

2.1 Hardware

The following page details the generic hardware setup involved in getting started with the ELM11.

2.1.1 Setup

The hardware setup is pretty straightforward, and can be distilled to just a single step:

1. Connect the hardware to your PC via USB-C.

2.1.2 Next

See [PC \(see page 10\)](#) to setup your PC such that it can communicate with the target hardware.

2.2 PC

The following page details configuring your PC such that it can communicate with the ELM11.

2.2.1 Setup

To establish a connection between your PC and the ELM11 follow the steps outlined below.

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 115200, after connecting the ELM11 to your PC.

```
user@pc> stty -F /dev/ttyUSB1 115200
```

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

2.2.1.2 Windows

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

2.2.2 Next

See [Connect!](#) (see page 12) to setup a connection between your PC and the ELM11.

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

2.3 Connect!

The following page details establishing a connection between your PC and the ELM11.

2.3.1 Setup

To establish a connection between your PC and the ELM11 follow the steps outlined below.

2.3.1.1 Linux or Mac OS

1. Launch your preferred serial terminal emulator. See [Serial Terminal Setup](#) (see page 16) 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](#)³ 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=**115200** bps, data bits=8, stop bits=1, parity=none and flow control=none. The recommended [tio](#)⁴ command line is as follows:

```
user@pc> sudo tio -b 115200 --output-delay 5 /dev/ttyUSB1
```

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

3. <https://ftdichip.com/document/installation-guides/>

4. <https://github.com/tio/tio>



On some flavours of Linux, we've found that the device must be forced to baud 115200, prior to establishing a serial connection (step 6).

```
user@pc> stty -F /dev/ttyUSB1 115200
```



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 115200 --output-delay 5 /dev/ttyUSB1 -s 2
```

2.3.1.2 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 the ELM11 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<ELM11 port number> and 'Speed' to 115200.



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.

2.3.2 Test

After powering on the board and completing step outlined above in the '**Setup**' section, you are now connected to the ELM11 in its default mode (**REPL Mode**). In this mode, the user is presented with a [Lua](#)⁵ **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 the hardware and software configuration metadata of the ELM11 processor) perform the following steps:

1. Unplug the ELM11 from your PC.
2. Close the serial terminal connection.
3. Press and hold the 'RST' button (see [01 \(ELM11 shipped default\)](#) (see page 53)).
4. Plug the ELM11 into your PC.
5. Re-establish serial terminal connection.
6. Release the 'RST' button.

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**.

2.3.3 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:

1. The program upload script, available [here](#)⁶.
2. A Python installation.

2.3.3.1 Python installation

2.3.3.1.1 Linux or Mac OS

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

5. <https://www.lua.org/>

6. https://brisbanesilicon.com.au/utilities/program_uploader.py


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

3. Install PySerial.

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

2.3.3.1.2 Windows

1. The Python installer for windows is available [here](#)⁷.
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
```

7. <https://www.python.org/downloads/windows/>

2.4 Serial Terminal Setup

The following page details information related to the setup of the PC terminal used to communicate with the ELM11.

2.4.1 Recommended Programs

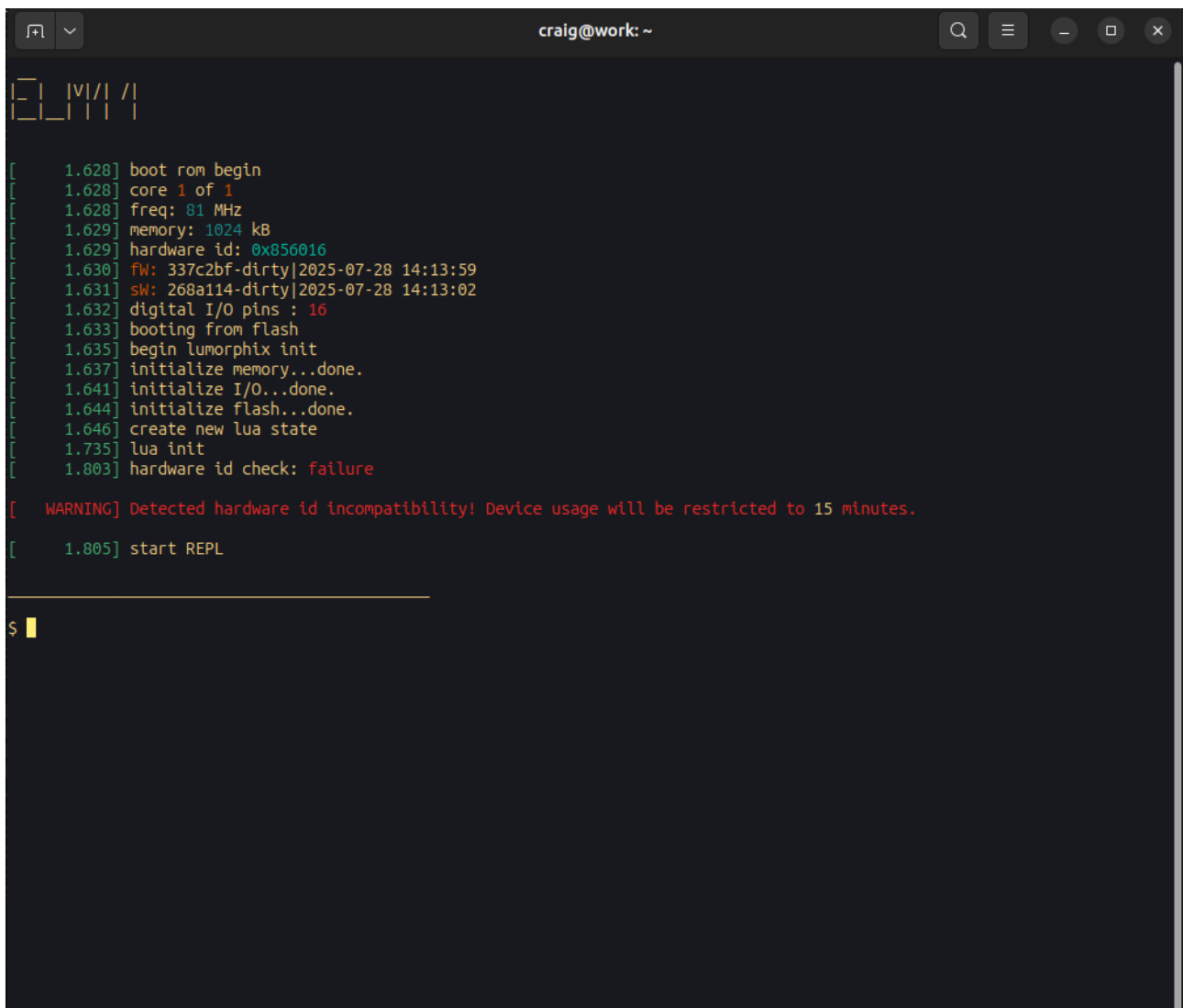
Of the various Serial Terminal emulators available, we recommend either [Tio](#)⁸ or [Putty](#)⁹.

2.4.2 Color Theme

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

8. <https://github.com/tio/tio>

9. <https://www.putty.org/>



A terminal window titled 'craig@work: ~' with a dark background and light-colored text. The window shows the boot process of an ELM11 device. The logs include hardware information, boot sequence steps, and a warning about hardware ID incompatibility. The terminal ends with a shell prompt '\$'.

```
[
  1.628] boot rom begin
[
  1.628] core 1 of 1
[
  1.628] freq: 81 MHz
[
  1.629] memory: 1024 kB
[
  1.629] hardware id: 0x856016
[
  1.630] fw: 337c2bf-dirty|2025-07-28 14:13:59
[
  1.631] sw: 268a114-dirty|2025-07-28 14:13:02
[
  1.632] digital I/O pins : 16
[
  1.633] booting from flash
[
  1.635] begin lumorphix init
[
  1.637] initialize memory...done.
[
  1.641] initialize I/O...done.
[
  1.644] initialize flash...done.
[
  1.646] create new lua state
[
  1.735] lua init
[
  1.803] hardware id check: failure

[ WARNING] Detected hardware id incompatibility! Device usage will be restricted to 15 minutes.

[
  1.805] start REPL

$
```

4 Dark Terminal Theme

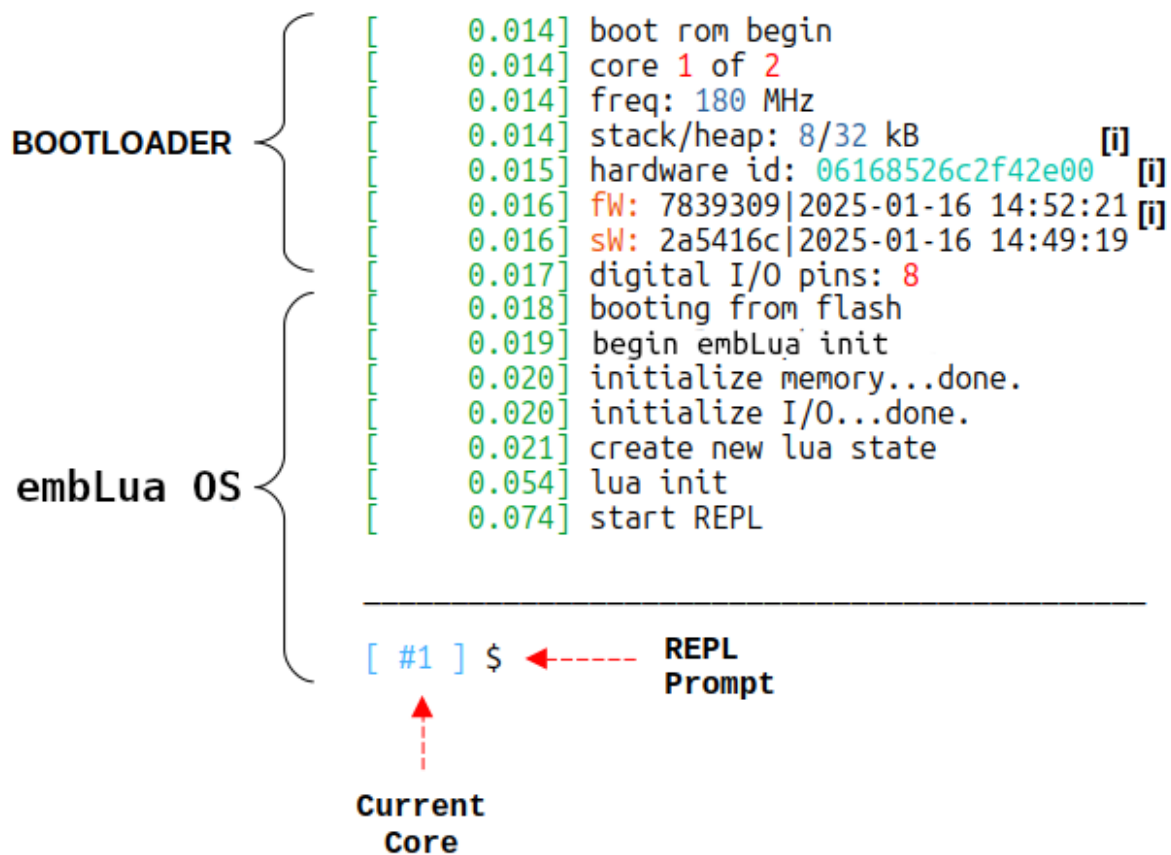
3 Boot Information

The following page details information related to the ELM11 boot process.

3.1 Boot Log Description

Upon power-on, the ELM11 will output a boot log via the User Comms channel (for the Beta release, see [Connect! \(see page 12\)](#)).

An image of the boot log, followed by a description, is as follows:



5 ELM11 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 ELM11 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 the embLua OS was built with doesn't match the physical hardware DNA, then a warning message will be printed (if the Current Core is #1) and the ELM11 will only run for fifteen minutes.

3.1.1 References

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

4 REPL Mode

The following page details the default ELM11 mode, the **REPL Mode**. If you are unfamiliar with a **REPL**, we recommend you familiarize yourself by reading some [background information](#)¹⁰.

4.1 Prompt

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

```
$
```

If the ELM11 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 ] $
```

4.2 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 ELM11 **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 ELM11 **REPL** fetched that variable and printed its value.
3. The variable named 'foo' is multiplied by itself, which equates to 15129.
4. A variable named 'foo_cubed' is declared, and given the value of foo * foo * foo.

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

- Finally, the user typed 'foo_cubed' and pressed return - the ELM11 **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 >999then
$$ print("Variable 'foo_cubed' is bigger than 999!")
$$ end
Variable 'foo_cubed' is bigger than 999!
```

4.3 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 ELM11 **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.

4.4 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 won't be stored in history that exceeds one hundred and twenty eight characters, in order to minimise memory utilization.

4.5 Summary

The above information provides a brief overview of the ELM11 **REPL**. Of course, in addition to supporting the Lua scripting language, the ELM11 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 addition to **REPL Mode**, the ELM11 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.

5 Command Mode

The following page details the ELM11 **Command Mode**.

5.1 Activation

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

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

5.2 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
```

5.3 Usage

5.3.1 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.

5.3.2 Supported Commands

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

5.3.2.1 List Commands

list|io_capabilities or **list|io_caps**

Lists the total I/O pins and capabilities of the current Core, for the current embLua OS 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 embLua OS 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 ELM11 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 embLua OS is being clocked at, in MHz.

list|program_count

Lists the number of programs currently hosted on the ELM11.

list|programs

Lists the programs currently hosted on the ELM11.

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).

5.3.2.2 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.

5.3.2.3 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'.

5.3.2.4 Delete Commands

delete|program("<Program Name>")

Delete the program "<Program Name>".

delete|all_programs

Delete all programs.

5.3.2.5 Load Commands

load|start_on_boot_io_type_cfg

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

5.3.2.6 Upload Commands

upload|program("<Program Name>")

Upload a program "<Program Name>" to the device. Requires the [program uploader](#)¹¹ 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 'USB1' accordingly):

```
$ python program_uploader.py <program name>.lua /dev/ttyUSB1 115200
```

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> 115200
```

5.3.2.7 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.

11. https://brisbanesilicon.com.au/utilities/program_uploader.py

run|reboot

Reboot the current Core.

5.3.2.8 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).

5.3.2.9 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.

5.3.2.10 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.

5.3.2.11 Other Commands

exit

Return to **REPL** mode.

6 API

The following section details the ELM11 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.

6.1 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:

6.1.1 Core

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

6.1.2 Pin

Constant	Description
PIN<x>	Represents I/O PIN number 'x', For example, in the REPL (or an uploaded program): <div> <pre>\$ print("Value of PIN7 is: "..PIN7) Value of PIN7 is: 7</pre> </div>
PIN<x>_BITMASK	Represents I/O PIN number 'x' bitmask. For example, handling interrupts: <div> <pre>\$ function interrupt(interrupt_vector) if (interrupt_vector & PIN3_BITMASK) ~= 0 then print("Interrupt occurred on PIN3!") end end</pre> </div>

6.1.3 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: <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <pre>\$ set_gpio(PIN7, TOGGLE)</pre> </div>

6.1.4 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	<p>Represents a I2C communications protocol, 'Digital I/O' pin type, with direction input and output. For example:</p> <pre>\$ set_io_config(PIN14, I2C)</pre>
------------	--

6.1.5 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	<p>Represents a high to low interrupt type on an I/O configured as GPIO_IN. For example, parsing interrupt types:</p> <pre>\$ 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</pre>

6.1.6 UART Interrupt

Constant	Description
----------	-------------

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

6.2 Functions

6.2.1 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.

6.2.1.1 Import

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

6.2.1.2 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	<p>Reset all I/O to type NONE (no arguments). For example:</p> <pre>\$ reset_all_io_type_cfg()</pre>

6.2.1.3 GPIO





Function	Description
set_gpio	<p>Set I/O (ARG1: PIN<x>), configured as GPIO_OUT, to level (ARG2: LOW, HIGH or TOGGLE). For example:</p> <pre>\$ set_gpio(PIN7, HIGH)</pre>
get_gpio	<p>Get I/O (ARG1: PIN<x>), configured as GPIO_IN, level. For example:</p> <pre>\$ 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!</pre>


6.2.1.4 PWM

Function	Description
set_pwm	<p>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).</p>


6.2.1.5 SPI


Function	Description
----------	-------------

spi_tx	<p>Same as 'spi_tx_byte'.</p> <div>  <p>The behavior in the case the ARG1 I/O PIN is configured to have a non byte-multiple data width:</p> <ul style="list-style-type: none"> • 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. </div>
spi_tx_byte	<p>Send 1 byte (ARG2: integer) of data to the I/O (ARG1: PIN<x>), configured as SPI_OUT.</p> <div>  <p>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'.</p> </div>
spi_tx_char	<p>Send 1 byte (ARG2: string) of data to I/O (ARG1: PIN<x>), configured as SPI_OUT.</p>
spi_tx_int	<p>Send 4 bytes (ARG2: integer) of data to I/O (ARG1: PIN<x>), configured as SPI_OUT.</p> <div>  <p>It is possible this will result in multiple SPI transmissions.</p> </div>
spi_rx_byte	<p>Receive one byte from I/O (ARG1: PIN<x>), configured as SPI_IN. This function will hang if no SPI data has been received.</p>
spi_rx_byte_nonblocking	<div>  <p>Not included in Beta Release.</p> </div>





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	<div>  Not included in Beta Release. </div>





6.2.1.6 UART

Function	Description
uart_tx	Same as 'uart_tx_byte'. For example: <div> <pre>\$ uart_tx(PIN4, 0x30)</pre> </div>
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 . <div>  This function will block if the corresponding UART module currently has no data, until a byte of data is received. </div>

uart_rx_byte_nonblocking	<p>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.</p> <div>  This function will not block - if the corresponding UART_IN module currently has no data then 'byte valid' will be false. </div>
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.

6.2.1.7 I2C

Function	Description
i2c_tx	<div>  Not included in Beta Release. </div>
i2c_tx_char	<div>  Not included in Beta Release. </div>
i2c_tx_byte	<div>  Not included in Beta Release. </div>
i2c_tx_int	<div>  Not included in Beta Release. </div>

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.

6.2.1.8 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_nonblocking	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_nonblocking	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).

6.2.1.9 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	<p>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).</p> <p>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).</p>
repl_ interrupt_ mode_disable	See above description.

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

interrupt_handler	<p>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:</p> <pre> \$ 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) </pre>
--------------------------	--

6.2.1.10 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.

6.2.1.11 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.

6.2.1.12 Core Communication

Function	Description
pipe_tx_byte	<p>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:</p> <pre> [#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 </pre>
pipe_tx_byte_nonblocking	<p>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.</p>

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.

6.2.1.13 Core Synchronization

Function	Description
lock	<p>Lock xbar barrier between Core<self> and Core <x> (ARG1: integer). Will block until barrier is unlocked by Core<x>.</p> <div>  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)). </div>
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.

6.2.1.14 Program

Function	Description
----------	-------------

run_program	<p>Run the program named (ARG1: string). For example:</p> <pre> \$ 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.</pre>
--------------------	---

6.2.1.15 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").**'

6.2.2 Interrupt Library

Function	Description
bind	<p>Bind the interrupt handler (ARG1: function) to the global interrupt source. Is part of the interrupt library, so must be called as such:</p> <pre>\$ interrupt.bind(interrupt_handler)</pre>
call	<p>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:</p> <pre>\$ interrupt.call()</pre>

6.2.3 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](#)¹²
- [table](#)¹³
- [math](#)¹⁴

12. <https://www.lua.org/pil/20.html>

13. <https://www.lua.org/pil/19.html>

14. <https://www.lua.org/pil/18.html>

6.3 Variables

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

Variable	Description
_VERSION	Lua version that embLua OS is currently running.
_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).
_cl	Current Core number and total Cores.
PWM_MAX	Maximum PWM duty cycle value (256).

7 Example Usage

The following section details examples of common usage of the ELM11.



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.

8 Hardware Overlays

The following section defines various hardware overlays that are available for the ELM11. New overlays are constantly being added, and can be requested via emailing: support@brisbanesilicon.com.au¹⁵.

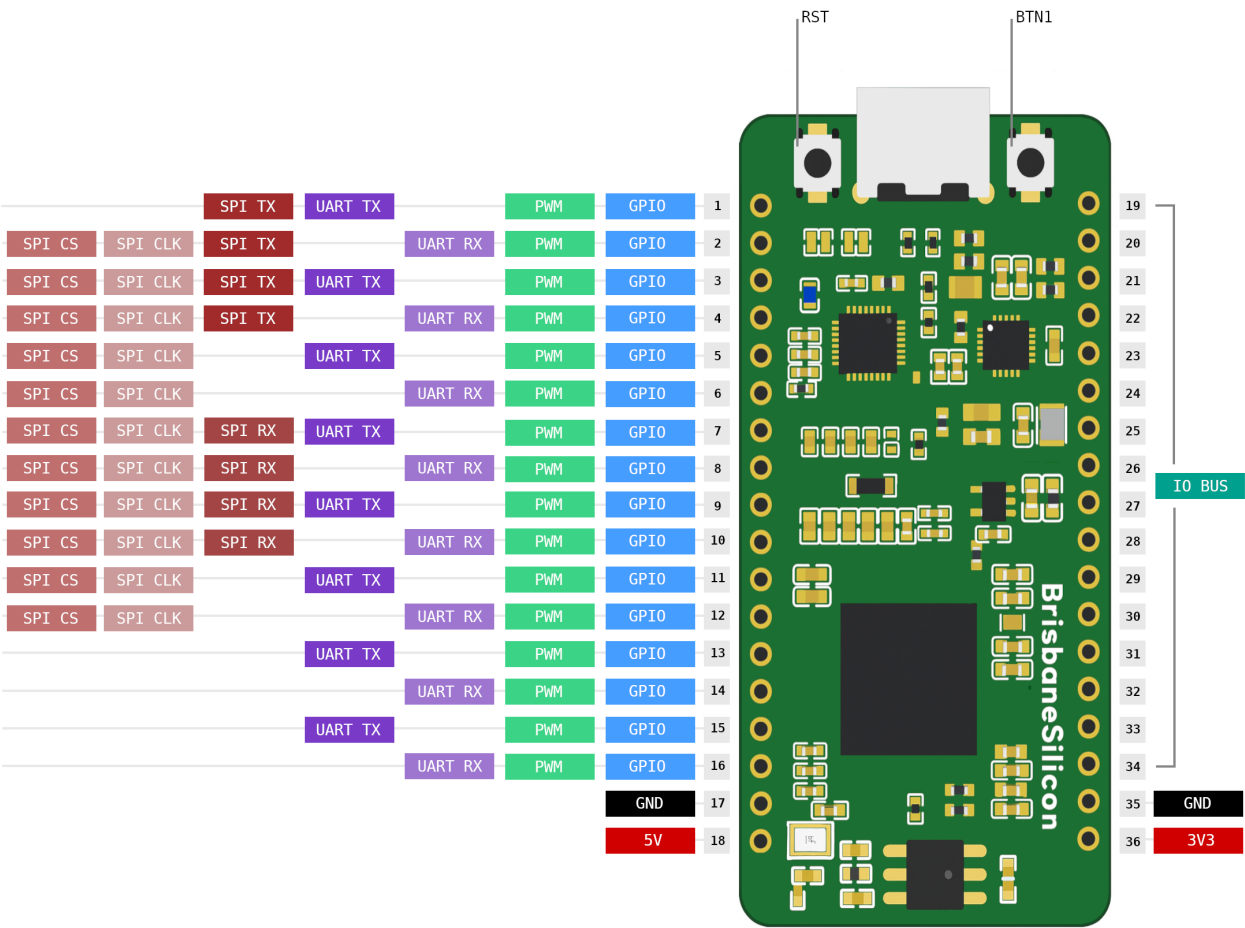
15. <mailto:support@brisbanesilicon.com.au>

8.1 01 (ELM11 shipped default)

8.1.1 CPU

Cores	1
Frequency	66 MHz
Heap	1 MB
Stack	40 KB
PINs	16
IO Bus	Write-only
VM Acceleration	No
Software Interrupts	All PINs
Hardware I/O Buffers	All PINs Skid-Buffer
UART Width	All PINs 8-bit
SPI Width	All PINs 8-bit
Hardware Watchdog	Yes
Performance Timer	Yes
General Timer	Yes

8.1.2 Pinout



6 Pinout of Hardware Overlay 01 (ELM11 shipped default)

8.2 02

Cores	1
Frequency	51 MHz
Heap	1 MB
Stack	40 KB
PINs	16
IO Bus	Write-only
VM Acceleration	No
Software Interrupts	No
Hardware I/O Buffers	No
UART Width	All PINs 8-bit
SPI Width	All PINs 8-bit
Hardware Watchdog	No
Performance Timer	No
General Timer	No

