elua Manual

v 0.01

Changelog:

2008 Nov, 30:

The present version of the eLua Manual is basically an assembly of the previous .txt documents.

Initial documentation for The "disp" module, with support for the OLED RIT128x96x4 display on the LM3S8962 Luminary Board was added.

Index

What is eLua?

eLua stands for **Embedded Lua** and the project aims to introduce the programming language Lua to the embedded world.

Lua is the perfect example of a minimal, yet fully functional language. Although generally advertised as a "scripting language" (and used accordingly especially in the game industry), it is also fully capable of running stand-alone programs. Its limited resource requirements make it suitable for a lot of microcontroller families. We began to use **eLua** on ARM microcontrollers, given their popularity, availability and small cost but support for other architectures are comming. Check the Status Page for updated info on supported platforms and features.

The aim of the project is to have a fully functional development environment **on the microcontroller itself**, without the need to install a specific toolchain on the PC side. Initially, a PC will still be needed in order to edit the Lua programs for the microcontroller. But as the project evolves this requirement will be relaxed, as a basic editor (also residing on the microcontroller) will be usable with a variety of input/output devices.

Authors

eLua is a joint project of **Bogdan Marinescu**, a software developer from Bucharest (Romania) and **Dado Sutter**, head of the Led Lab at PUC-Rio University, in Rio de Janeiro (Brazil).

Its origins come from the ReVaLuaTe project also developed by Bogdan Marinescu (as a contest entry for the 2005 Renesas M16CDesign Contest), and the Volta Project, managed by Dado Sutter at PUC-Rio from 2005 to 2007.

eLua is an Open Source and colaborative project and thus it's code has worldwide contributors.

Licence

eLua is Open Source and is freely distributed under the GPL (migrating to BSD soon) licence.

The Lua code (with slight modifications) is included in the source tree, and its of course licensed under the same MIT license that Lua uses.

The terms of these licences can be viewed on their pages at:

GPL Licence

BSD Licence

MIT Licence

Building eLua

Up do date documentation of how to build eLua is always included in the eLua distribution in the docs directory (docs/building.txt). For your convenience, the building instructions are also provided on this page.

Prerequisites

Before you start, you might want to check if the list of platform modules and eLua components are set according to your needs. See docs/platform_modules.txt and docs/elua_components.txt for details.

Building eLua

To build eLua you'll need:

- a GCC/Newlib toolchain for your target (see http://elua.berlios.de for instructions on how to build your own toolchain). Please note that even if you already have a compiled toolchain, the differences in the Newlib configure flags (mainly the --disable-newlib-supplied-syscalls flags) might prevent eLua for building properly on your machine.
- Linux. Compiling under windows should be possible, however this isn't tested. I'm using Ubuntu, so I'm also using "apt-get". If you're using a distro with a different package manager you'll need to translate the "apt-get" calls to your specific distribution.
- python. It should be already installed; if it's not:

  $ sudo apt-get install python
- scons. eLua uses scons instead of make and makefiles, because I find scons much more "natural" and easier to use than make. To install it:

  $ sudo apt-get install scons
- your toolchain's "bin" directory (this is generally something like /usr/local/cross-arm/bin, where /usr/local/cross-arm is the directory in which you installed your toolchain) must be in $PATH.
- if you're building for the i386 platform, you'll also need "nasm":

$ sudo apt-get install nasm

For each platform, eLua assumes a certain name for the compiler/linker/assembler executable files, as shown below.

```
================================================================================
| Tool        |      Compiler        |        Linker         |     Assembler      |
|-------------|----------------------|-----------------------|--------------------|
| Platform    |                      |                       |                    |
|=============|======================|=======================|====================|
| ARM (all)   |    arm-elf-gcc       |      arm-elf-gcc      |     arm-elf-gcc    |
|=============|======================|=======================|====================|
| i386        |    i686-elf-gcc      |      i686-elf-gcc     |        nasm        |
|=============|======================|=======================|====================|
| Cortex-M3   |    arm-elf-gcc       |      arm-elf-gcc      |     arm-elf-gcc    |
|=============|======================|=======================|====================|
```

If your toolchain uses different names, you have to modify the "conf.py" file from src/platform/[your platform].

To build, go to the directory where you unpacked your eLua distribution and invoke scons:

```
$ scons [target=lua | lualong]
  [cpu=at91sam7x256 | at91sam7x512 | i386 | str912fw44 | lm3s8962 |
        lm3s6965 | lpc2888 | str711fr2 ]
  [board=ek-lm3s8962 | ek-lm3s6965 | str9-comstick | sam7-ex256 | lpc-h2888 |
        | mod711 | pc]
  [cpumode=arm | thumb]
  [allocator = newlib | multiple]
  [prog]
```

Your build target is specified by two paramters: cpu and board. "cpu" gives the name of your CPU, and "board" the name of the board. A board can be associated with more than one CPU. This allows the build system to be very flexible. You can use these two options together or separately, as shown below:

- cpu=name: build for the specified CPU. A board name will be assigned by the build system automatically.
- board=name: build for the specified board. The CPU name will be inferred by the build system automatically.
- cpu=name board=name: build for the specified board and CPU.

For board/CPU assignment look at the beginning of the SConstruct file from the base directory, it's self-explanatory.

The other options are as follows:

- target=lua | lualong: specify if you want to build full Lua (with floating point support) or integer only Lua (lualong). The default is "lua".
- cpumode=arm | thumb: for ARM target (not Cortex) this specifies the compilation mode. Its default value is 'thumb' for AT91SAM7X targets and 'arm' for STR9 and LPC2888 targets.
- allocator = newlib | multiple: choose between the default newlib allocator (newlib) and the multiple memory spaces allocator (multiple). You should use the 'multiple' allocator only if you need to support multiple memory spaces, as it's larger that the default Newlib allocator (newlib). For more information about this reffer to platform_interface.txt. The default value is 'newlib' for all CPUs except 'lpc2888', since my lpc-h2888 comes with external SDRAM memory and thus it's an ideal target for 'multiple'.
- prog: by default, the above 'scons' command will build only the 'elf' file. Specify "prog" to build also the platform-specific programming file where appropriate (for example, on a AT91SAM7X256 this results in a .bin file that can be programmed in the CPU).

The output will be a file named elua*[target]*[cpu].elf (and also another file with the same name but ending in .bin if "prog" was specified for platforms that need .bin files for programming). If you want the equivalent of a "make clean", invoke "scons" as shown above, but add a "-c" at the end of the command line. "scons -c" is also recommended after you change the list of modules/components to build for your target (see section "prerequisites" of this document), as scons seems to "overlook" the changes to these files on some occasions.

A few examples:

```
$ scons cpu=at91sam7x256
```

Build eLua for the AT91SAM7X256 CPU. The board name is detected as sam7-ex256.

```
$ scons board=sam7-ex256
```

Build eLua for the SAM7-EX256 board. The CPU is detected as AT91SAM7X256.

```
$ scons board=sam7-ex256 cpu=at91sam7x512
```

Build eLua for the SAM7-EX256 board, but "overwrite" the default CPU. This is useful when you'd like to see how the specified board would behave with a different CPU (in the case of the SAM7-EX256 board it's possible to switch the on-board AT91SAM7X256 CPU for an AT91SAM7X512 which has the same pinout but comes with more Flash/RAM memory).

```
$ scons cpu=lpc2888 prog
```

Build eLua for the lpc2888 CPU. The board name is detected as LPC-H2888. Also, the bin file required for target programming is generated.

Using eLua

So, you already built and installed eLua, but now you don't know what to do with it. It's actually quite easy: all you need is your board connected to the computer and a terminal emulation program. If you're using Windows, I strongly recommend TeraTerm. It's a freeware, it's very powerful and also easy to use. On Linux, you'll probably be stucked with minicom. It's not exactly intuitive, and it runs in text mode, but it's still very powerful, and if you google for "minicom tutorial" you'll get the hang of it in no time. Or you can try any other terminal emulator, as long as you set it up properly (and as long as it gives you the option of transferring files via XMODEM, which is what eLua uses at the moment). These are the main settings you need to look at:

- port setup: 115200 baud (38400 for STR7), 8N1(8 data bits, no parity, one stop bit).
- flow control: none
- newline handling: "CR" on receive, "CR+LF" on send (some terminal programs won't give you a choice here).

Also, depending on the type of your board, you'll need some way to connect the board to a serial port on your PC, or to USB if you're using an USB to serial converter. For example, as already explained here, the USB port on the LM3S7862 board is dual, so you can use it as an USB to serial converter after downloading your firmware, thus you don't need any other type of connection. The same is true for the STR9-comStick board. On the other hand, for the SAM7-EX256 board you'll need to connect a serial cable to the "RS232" connector, provided that the jumpers are already set as explained here.

After you press the "RESET" button on your board, you should see the eLua shell prompt. Up to date documentation of how to use the shell is always included in the distribution (docs/the_elua_shell.txt). For your convenience, the shell documentation is also provided on this page.

The eLua shell

After you burn eLua to your board and you connect the board to your terminal emulator running on the PC, you'll be greeted with the eLua shell prompt, which allows you to:

- run 'lua' as you would run it from the Linux or Windows command prompt
- upload a Lua source file via XMODEM and execute in on board
- query the eLua version
- get help on shell usage

To enable the shell, define BUILD_SHELL in your build.h file, and also BUILD_XMODEM if you want to use the "recv" command (see below). See docs/elua_components.txt for more details about enabling the shell.

You'll need to configure your terminal emulation program to connect to your eLua board. These are the parameters you'll need to set for your serial connection:

- speed 115200, 8N1 (8 data bits, no parity, one stop bit)
- no flow control
- newline handling (if available): CR on receive, CR+LF on send

After you setup your terminal program, press the RESET button on the bord. When you see the "eLua# " prompt, just enter "help" to see the on-line shell help:

eLua# help
Shell commands:

help - print this help
lua [args] - run Lua with the given arguments
recv - receive a file (XMODEM) and execute it
ver - print eLua version
exit - exit from this shelll

More details about some of the shell commands are presented below.

The "recv" command

To use this, your eLua taret image must be built with support for XMODEM (see docs/elua_components.txt for details). Also, your terminal emulation program must support sending files via the XMODEM protocol. Both XMODEM with checksum (the original version) and XMODEM with CRC are supported, but only XMODEM with 128 byte packets is allowed (XMODEM with 1K packets won't work). To use this feature, enter "recv" at the shell prompt. eLua will respond with "Waiting for file ...". At this point you can send the file to the eLua board via XMODEM. eLua will receive and execute the file. Don't worry when you see 'C' characters suddenly appearing on your terminal after you enter this command, this is how the XMODEM transfer is initiated.

The "lua" command

This allows you to start the Lua interpreter with command line parameters, just as you would do from a Linux or Windows command prompt. This command has some restrictions:

- the command line can't be longer than 50 chars
- character escaping is not implemented. For example, the next command won't work because of the ' escape sequences:

eLua# lua -e 'print('Hello, World!')' -i
Press CTRL+Z to exit Lua
lua: (command line):1: unexpected symbol near ''

However, if you use both '' and "" for string quoting, it will work:

eLua# lua -e 'print("Hello, World")' -i
Press CTRL+Z to exit Lua
Lua 5.1.4 Copyright (C) 1994-2008 Lua.org, PUC-Rio Hello,World

eLua on LM3S CPUs

Using eLua with the LM3S (Cortex-M3) CPUs from Luminary Micro

Luminary Micro is the company that produced the world's first silicon implementation of the Cortex-M3 processor. Their device portfolio is quite impressive, ranging from relatively simple devices to full-featured CPUs (with on-chip USB, EMAC, CAN, and many other peripherals). The support package for these devices is also very good, with drivers for all the CPU peripherals and ports of 3rd party applications. And, on a personal note, I contacted Luminary Micro some while ago with a request to support this project with one of their evaluation kits, and their response was excellent (thanks again, Luminary!). That's how a LM3S8962-EK landed on my desk. This is the development board that I'm going to use in this tutorial (of course you might still try this if you have a different LM3S8962 board).

NOTE: Starting with version 0.3, eLua has support for the LM3S6965 CPU. All the instructions in this tutorial are applicable to the LM3S6965-EK with minimal changes.

Prerequisites

Before you'll be able to use eLua on the LM3S8962 CPU, make sure that:

- you're using Windows. Yes, I really said Windows. The reason is quite simple: we're going to use Luminary's tools to burn eLua to the EK, and they're Windows specific. This is the case with many CPUs and vendors out there, so get used to the idea. You can have Windows installed on your HDD, or under an emulator in Linux, it doesn't matter, you can even try to run it from Wine if you're really, really brave. I'm using XP, Vista should work too.
- you have installed the LM Flash Programmer tool from Luminary. Look for it on this page, for example (the link is in the "Software updates" table).
- you already built your eLua image for the LM3S8962 CPU. Simply put, this means that you have a GCC toolchain for Cortex-M3, and that you used it to build eLua (remember to specify "prog" on the scons command line to get a .bin file that's suitable for programming). Or follow the instructions from the download page and download a precompiled binary image.

Burning eLua on the LM3S8962-EK

Fortunately, this is as easy and painless as possible. One of the nicest things about the LM3S8962-EK is that it uses the on-board USB port for both firmware downloading and for emulating a serial port (via a hardware USB to UART converter, so you don't need any special software on the CPU to access this UART port). Moreover, it automagically knows how (and when) to switch from the firmware download mode to the UART emulation mode, so you don't need to move jumpers around or anything like this. It's zero effort firmware upgrading at its best. So, let's do it:

- connect your board to your PC using a suitable USB cable. If you didn't install the board drivers yet, you'll be asked to install them now.
- if you're already using the USB connection on the board in the UART emulation mode, close your terminal program (or at least disconnect it from the USB COM port).
- fire up the "Luminary Micro Flash Programmer" application.
- in the "Configuration" tab, select "LM3S8962 Ethernet and CAN Evaluation board".
- in the "Program" tab, select the eLua .bin file that you got from the compilation step.
- select the "Options" as you like (I generally choose "Erase entire flash" and Reset MCU after

program").

- hit the "Program" button.
- wait until programming is over, then exit the flash programmer application.

That's it! eLua is now programmed in the CPU, so you can start your terminal emulator and enjoy it, as described in using eLua. If you have any problems with this procedure, feel free to contact me. Although, if you know how to burn the image from Linux, please let me know and I'll include the instructions in this page. Since the on-board programming interface in still JTAG, this can surely be done with a JTAG tool like OpenOCD, but I don't know much about such tools.

eLua on AT91SAM CPUs

Using eLua with the AT91SAM7X CPUs from Atmel

Atmel is a company that doesn't need any kind of introduction from me :) Their huge product range include some quite nice ARM7TDMI core implementations. Among them are the AT91SAM7X256 and AT91SAM7X512 CPUs. The only difference between them is the ammount of internal memory (256k Flash+64k RAM for AT91SAM7X256 vs. 512k Flash+128k RAM for AT91SAM7X512). Loaded with peripherals, and accompanied by a good support package, they make a perfect host for eLua. For this tutorial I'm going to use the SAM7-EX256 development board from Olimex. It's quite a decent board, and also reasonably priced, although it lacks a proper documentation package in my oppinion. It is equipped with an AT91SAM7X256 CPU. As much as I'd like to get my hands on a board with a AT91SAM7X512 CPU, this didn't happen so far, so I'm going to stick with AT91SAM7X256. Of course, you can still try this tutorial if you have a different AT91SAM7X256 development board. Plus, the instructions should be quite similar for AT91SAM7X512 CPUs.

Prerequisites

Before you'll be able to use eLua on the AT91SAM7X256 CPU, make sure that:

- you're using Windows. Well, this is debatable. Unlike the LM3S CPU, the Atmel CPU is supported by the excellent OpenOCD package, so programming it from Linux is definitely possible, as OpenOCD runs equally well on Windows and Linux. However, since I'm forced to use Windows anyway because of the restrictions of some of my other development boards, I'm going to take advantage of this and cover the Atmel programming tool instead of OpenOCD. The advantage is that you don't need a JTAG "dongle" to program your board (which would be the case if you were using OpenOCD). The disadvantage, of course, is that the Atmel tool runs only on Windows. Plus, I personally find OpenOCD tedious to use. If you still want to use it though, you might want to check the forementioned Olimex page, they have some OpenOCD related links there. That said, from now on I'm going to assume that you use Windows. I'm using XP, Vista should work too.
- you have installed the AT91 In-system Programmer (ISP) package from Atmel.
- you already built your eLua image for the AT91SAM7X256 CPU. Simply put, this means that you have a GCC toolchain for ARM, and that you used it to build eLua (remember to specify "prog" on the scons command line to get a .bin file that's suitable for programming). Or follow the instructions from the download page and download a precompiled binary image.

Burning eLua on the SAM7-EX256 board

This involves some jumper tricks, but it's still easy enough to do. We'll need to play with four jumpers: the "USB/EXT" jumper (located to the right of the USB connector from the bottom left part of the board in its close proximity), the "ERASE" jumper (located at the right of the "UEXT" header connector in the top-left part of the board, right ahead the quartz), and the block of two jumpers located right under the "RS232" connector on the board (the one that is adjacent to the Ethernet connector on its right side, NOT the one labeled "CAN" that is closer to the right edge of the board).

- connect your board to your PC using a suitable USB cable.
- if you have a terminal emulation program connected to the board, close it (or at least disconnect

it from its port).

- make sure that the the block of two jumpers mentioned before is set to positions "RXD0" and "TXD0" respectively, NOT "DRXD" and "DTXD".
- make sure that the "USB/EXT" connector is set to "USB" (position 1-2) and that the "ERASE" jumper is disconnected.
- connect the "ERASE" jumper and wait one second or more.
- disconnect the "USB/EXT" jumper completely, then disconnect the "ERASE" jumper too.
- connect the "USB/EXT" jumper back in the "USB" position (1-2).
- fire up the Atmel programming tool. If you haven't installed your board yet, you'll be asked to do so at this point.
- select "\usb\ARMx" as the connection (for me it's \usb\ARM0) and "AT91SAM7X256-EK" as the board.
- select the "Flash" tab from the middle tab of the window.
- in the "Send file name" box select your eLua bin file that you got from the compilation step and then press "Send File".
- wait for the file to be sent and answer "No" to the "Lock region(s)" dialog.
- in the window section below ("Scripts") select "Boot from Flash (GPNVM2)" then press "Execute".
- exit the application.

Phew! That was no walk in the park, but at least eLua is now programmed in the CPU, so you can start your terminal emulator and enjoy it, as described in using eLua. If you have any problems with this procedure, feel free to contact me. Although, if you know how to burn the image from Linux/Windows with OpenOCD, please let me know and I'll include the instructions in this page.

eLua on STR9 CPUs

Using eLua with the STR9 CPUs from ST

Among the ARM based MCUs available today, the STR9 CPUs from ST stand up because of a few unique features. First, their core is an ARM966-E, as opposed to the very popular ARM7TDMI core. This, together with some cleverly chosen on-chip hardware blocks, allows the CPU to run at 96MHz, which is very fast for a general purpose MCU. The particular CPU I'm using (STR912FW44)) also has 512k of flash (and another bank of 32k flash) and 96k of internal RAM, so you won't be running out of memory anytime soon. It is accompanied by a very good support library, and ST provides a lot of nice tools for STR9, including a graphical tool that you can use to configure the chip exactly how you want. When I wrote to ST about eLua, they agreed to send me a STR9-comStick board to run eLua on it. Thank you very much for your help, once again. This is the board that I'm going to use through this tutorial.

Prerequisites

Before you'll be able to use eLua on the STR912FW44 CPU, make sure that:

- you're using Linux, Windows, or any other OS that has support for OpenOCD. You might have a look at my OpenOCD tutorial before continuing.
- if you're on Windows, you have installed the STR9-comStick support package from the accompanying CD.
- you already built your eLua image for the STR912FW44 CPU. Simply put, this means that you have a GCC toolchain for ARM, and that you used it to build eLua (remember to specify "prog" on the scons command line to get a .bin file that's suitable for programming). Or follow the instructions from the download page and download a precompiled binary image.

Burning eLua to the STR9-comStick

You need OpenOCD to do this. Just follow the instructions from my OpenOCD tutorial . On the tutorial page you'll also find links to the OpenOCD configuration files that I'm using for burning eLua to the comStick.

IMPORTANT NOTE: for some very strage reasons (probably related to the on-board USB to JTAG converter) my comstick does NOT start to execute the code from its internal flash after being powered up via the USB cable (faulty reset sequence?). To overcome this, you'll find a special OpenOCD configuration file on my OpenOCD tutorial page. It is called comrst.cfg, and you can use it to reset your comstick after it is powered up.

That's it! eLua is now programmed in the CPU, so you can start your terminal emulator and enjoy it, as described in using eLua. If you have any problems with this procedure, feel free to contact us.

eLua on STR7 CPUs

Using eLua with the STR7 CPUs from ST

STR7 is a family of ATM7TDMI based CPUs from ST. They are small, low power MCUs, with a well balanced set of on-chip peripherals. I'm using the MOD711 header board from ScTec. The board is based on this STR711FR2 variant of the STR7 family. Since this is not a full-fledged development board, I had to add a few things around it: a MAX3232 RS232 to TTL converter for the serial interface, a couple of LEDs and a reset button. After that, the board was ready for some eLua :)

Prerequisites

Before you'll be able to use eLua on the STR711FR2 CPU, make sure that:

- you're using Linux, Windows, or any other OS that has support for OpenOCD. You might have a look at my OpenOCD tutorial before continuing.
- you already built your eLua image for the STR711FR2 CPU. Simply put, this means that you have a GCC toolchain for ARM, and that you used it to build eLua (remember to specify "prog" on the scons command line to get a .bin file that's suitable for programming). Or follow the instructions from the download page and download a precompiled binary image.

Burning eLua to the MOD711 board

You need OpenOCD to do this. Just follow the instructions from my OpenOCD tutorial . On the tutorial page you'll also find links to the OpenOCD configuration files that I'm using for burning eLua to the MOD711 board. And that's it! eLua is now programmed in the CPU, so you can start your terminal emulator and enjoy it, as described in using eLua. **IMPORTANT NOTE**: for this board you need to set your COM port speed to 38400 baud (as opposed to 115200 baud for the other boards). All the other parameters are the same (8 data bits, no parity, one stop bit). If you have any problems with this procedure, feel free to contact us.

eLua on LPC2888 CPUs

Using eLua with the LPC2888 CPU from NXP

The LPC2888 CPU from NXP packs some interesting features: huge internal 1Mbyte flash memory, on-chip USB 2.0 high speed interface, and the most complex (by far) clocking network that I've ever seen on an ATM7TDMI chip. Also, it implements the USB DFU (Device Firmware Update) profile over its USB interface, so it's quite easy to program it in-circuit. I'm using the Olimex LPC-H2888 development board built around this chip, which packs 32MBytes of external SDRAM and also 2MBytes of external flash, which is more than enough for my needs. However, it does have its fair share of downsides. For starters, its support package (from NXP) is very poot when compared to other targets on which eLua runs. You don't even get drivers for all your peripherals, just a a few (quite incomplete) examples. Its datasheet could be much more explit at times, especially when referring to the clocking section (which is quite complicated). On my board, the DFU download mode (firmware upgrade via USB) stopped working out of the blue, without any apparent reasons, and I was unable to use DFU on the chip since then, I had to resort to using OpenOCD (and come up with a configuration file, since it was impossible to find one for LPC2888). The CPU itself has a very interesting limitation: because of a sillicon error, it's impossible to run Thumb code from the on-chip flash, you can only run regular ARM code (?!). Also, the board that I got from Olimex completely ignores the fact that this chip can run in DFU mode (it doesn't include any kind of jumper and/or switch to enable this mode), so I had to build a support board for it. Which is something I had to do also because the board doesn't export a RS232 interface, I had to build one around a MAX232 chip. All in all, my experience with this chip (and with the Olimex board) wasn't that pleasant, but this doesn't change the fact that the LPC-H2888 is the most powerful (resource-wise) board on which eLua runs.

Prerequisites

Before you'll be able to use eLua on the LPC2888 CPU, make sure that:

- if you're going to use DFU for firmware programming, you'll need Windows (although I heard reports of Linux programs that can program this chip in DFU mode, but I won't cover them here). If you're going to use OpenOCD, Linux, Windows, or any other OS that has support for OpenOCD will do. In this case, you might want to have a look at my OpenOCD tutorial before continuing.
- also, if you're going to use DFU, you'll need a way to boot the chip in DFU firmware upgrade mode. This is done by pulling up (tie to VCC) the P2.3 pin at startup. On my board I included a switch for this. Press the switch, press RESET while holding the switch pressed, then release the switch. You chip is now in DFU mode.
- if you're using DFU, you have installed the LPC2888 flash programming utility from here (the package also contains the Windows DFU drivers).
- if you're using OpenOCD, you have followed the instructions from my OpenOCD tutorial.
- you already built your eLua image for the LPC2888 CPU. Simply put, this means that you have a GCC toolchain for ARM, and that you used it to build eLua (remember to specify "prog" on the scons command line to get a .bin file that's suitable for programming). Or follow the instructions from the download page and download a precompiled binary image.

Burning eLua to the LPC2888 using the DFU tool from NXP

The DFU flashing application doesn't work directly on the .bin files you get after building eLua, you need to run them though NXP's "hostcrypt" program (which is part of the LPC2888 DFU package). After you have your eLua .bin file, do this from a Windows command prompt (make sure that hostcryptv2.exe is in the path):

```
C:\> hostcryptv2 elua_lua_lpc2888.bin elua.ebn -K0 -F0
```

As a result, you'll have a new file (elua.ebn). Now boot your chip in DFU firmware upgrade mode (see above) and use the DFU utility (MassDFUApplication.exe) to load elua.ebn into your chip (the instructions on using MassDFUApplication are in a PDF file that's included in the LPC2888 DFU package). Reset the board and enjoy.

Burning eLua to the LPC2888 using OpenOCD

If you're as lucky as me and your board refuses to use DFU anymore, follow our OpenOCD tutorial to burn your image using OpenOCD.

eLua on on i386 CPUs

Using eLua with Intel i386 (or better) CPUs

Since the i386 platform was implemented as a proof of concept only, the only things you can do with it are:

Boot your PC in eLua (from a Hard Disk)

Boot eLua from a memory stick / pen drive

If you want to do this, build your eLua image or download a precompiled image, as explained in the download page. However, most of the features that you'd find on an embedded platform won't work. You won't be able to upload programs to your i386 eLua box using the XMODEM protocol (not because it's impossible, but simply because this doesn't make sense at all on a desktop PC). Also, you won't be able to control the peripherals that you'd normally find in an embedded CPU (SPI, I2C, PIO and all the others), because they are not present on the i386 platform (they can be emulated via different means, but this is way beyond the scope of eLua). So, until further notice, i386 will be nothing more than a spectacular demo platform for eLua. If you think that you can make something more out of it, please feel free to contact me. I'm actually very interested in this, but I lack the necessary resources to continue it.

Tutorials

Building GCC for ARM

This tutorial explains how you can create a GCC+Newlib toolchain that can be used to compile programs for the ARM architecture, thus making it possible to compile programs for the large number of ARM CPUs out there. You'll need such a toolchain if you want to compile eLua for ARM CPUs. This tutorial is similar to many others you'll find on the Internet (particulary the one from gnuarm, on which it's based), but it's a bit more detailed and shows some "tricks" you can use when compiling Newlib.

DISCLAIMER: I'm by no means a specialist in the GCC/newlib/binutils compilation process. I'm sure that there are better ways to accomplish what I'm describing here, however I just wanted a quick and dirty way to build a toolchain, I have no intention in becoming too intimate with the build process. If you think that what I did is wrong, innacurate, or simply outrageously ugly, feel free to contact me and I'll make the necessary corrections. And of course, this tutorial comes without any guarantees whatsoever.

› Prerequisites

To build your toolchain you'll need:

- a computer running Linux: I use Ubuntu 8.04, but any Linux will do as long as you know how to find the equivalent of "apt-get" for your distribution. I won't be going into details about this, google it and you'll sure find what you need. It is also assumed that the Linux system already has a "basic" native toolchain installed (gcc/make and related).This is true for Ubuntu after installation. Again, you might need to check your specific distribution.
- GNU binutils: get it from here. At the moment of writing this, the latest versions is 2.18, which for some weird reason refuses to compile on my system, so I'm using 2.17 instead.
- GCC: version 4.3.0 or newer is recommended. As I'm writing this, the latest GCC version is 4.3.1 which I'll be using for this tutorial. Download it from here after choosing a suitable mirror.
- Newlib: as I'm writing this, the latest official Newlib version is 1.16.0. Download it from the Newlib FTP directory.
- Also, the tutorial assumes that you're using bash as your shell. If you use something else, you might need to adjust some shell-specific commands.

Also, you need some support programs/libraries in order to compile the toolchain. To install them:

$ sudo apt-get install flex bison libgmp3-dev libmpfr-dev autoconf texinfo

Next, decide where you want to install your toolchain. They generally go in /usr/local/, so I'm going to assume /usr/local/cross-arm for this tutorial. To save yourself some typing, set this path into a shell variable:

$ export TOOLPATH=/usr/local/cross-arm

› Step 1: binutils

This is the easiest step: unpack, configure, build.

$ tar xvfj binutils-2.17.tar.bz2

$ cd binutils-2.17

$ mkdir build

$ cd build

$ ../configure --target=arm-elf --prefix=$TOOLPATH --enable-interwork --enable-multilib--with-gnu-as --with-gnu-ld --disable-nls

$ make all

$ sudo make install

$ export PATH=${TOOLPATH}/bin:$PATH

Now you have your ARM "binutils" (assembler, linker, disassembler ...) in your PATH.

› Step 2: basic GCC

In this step we build a "basic" GCC (that is, a GCC without any support libs, which we'll use in order to build all the libraries for our target). But first we need to make a slight modification in the configuration files. Out of the box, the GCC 4.3.1/newlib combo won't compile properly, giving a very weird "Link tests are not allowed after GCC_NO_EXECUTABLES" error. After a bit of googling, I found the solution for this:

$ tar xvfj gcc-4.3.1.tar.bz2

$ cd gcc-4.3.1/libstdc++-v3

$ joe configure.ac

I'm using "joe" here as it's my favourite Linux text mode editor, you can use any other text editor. Now find the line which says "AC_LIBTOOL_DLOPEN" and comment it out by adding a "#" before it:

```
# AC_LIBTOOL_DLOPEN
```

Save the modified file and exit the text editor

$ autoconf

$ cd ..

Great, now we know it will compile, so let's do it:

$ mkdir build

$ cd build

$ ../configure --target=arm-elf --prefix=$TOOLPATH --enable-interwork --enable-multilib --enable-languages="c,c++" --with-newlib --without-headers --disable-shared--with-gnu-as --with-gnu-ld

$ make all-gcc

$ sudo make install-gcc

On my system, the last line above (sudo make install-gcc) terminated with errors, because it was unable to find our newly compiled binutils. If this happens for any kind of "make install" command, this is a quick way to solve it:

```
$ sudo -s -H
# export PATH=/usr/local/cross-arm/bin:$PATH
# make install-gcc
# exit
```

› Step 3: Newlib

Once again, Newlib is as easy as unpack, configure, build. But I wanted my library to be as small as possible (as opposed to as fast as possible) and I only wanted to keep what's needed from it in the final executable, so I added the "-ffunction-sections -fdata-sections" flags to allow the linker to perform dead code stripping:

```
$ tar xvfz newlib-1.16.0.tar.gz
$ cd newlib-1.16.0
$ mkdir build
$ cd build
$ ../configure --target=arm-elf --prefix=$TOOLPATH --enable-interwork --disable-newlib-supplied-syscalls --with-gnu-ld --with-gnu-as --disable-shared
$ make CFLAGS_FOR_TARGET="-ffunction-sections -fdata-sections -DPREFER_SIZE_OVER_SPEED -D__OPTIMIZE_SIZE__ -Os -fomit-frame-pointer -D__BUFSIZ__=256"
$ sudo make install
```

Some notes about the flags used in the above sequence:

- `--disable-newlib-supplied-syscalls`: this deserves a page of its own, but I won't cover it here. For an explanation, see for example this page
- `-DPREFER_SIZE_OVER_SPEED -D__OPTIMIZE_SIZE__`: compile Newlib for size, not for speed (these are Newlib specific).
- `-Os -fomit-frame-pointer`: tell GCC to optimize for size, not for speed.
- `-D__BUFSIZ__=256`: again Newlib specific, this is the buffer size allocated by default for files opened via fopen(). The default is 1024, which I find too much for an eLua, so I'm using 256 here. Of course, you can change this value.

› Step 4: full GCC

Finally, in the last step of our tutorial, we complete the GCC build. In this stage, a number of compiler support libraries are built (most notably libgcc.a). Fortunately this is simpler that the Newlib compilation step:

```
$ cd gcc-4.3.1/build
$ make all
$ sudo make install
```

› Step 5: all done!

Now you can finally enjoy your ARM toolchain, and compile eLua with it :) If you need further clarification, or if the above instructions didn't work for you, feel free to contact me.

Building GCC for Cortex

This tutorial explains how you can create a GCC+Newlib toolchain that can be used to compile programs for the Cortex (Thumb2) architecture, thus making it possible to use GCC to compile programs for the increasingly number of Cortex CPUs out there (Luminary Micro, ST, with new Cortex CPUs being announced by Atmel and other companies). I am writing this tutorial because I needed to work on a Cortex CPU for the eLua project and I couldn't find anywhere a complete set of instructions for building GCC for this architecture. You'll need such a toolchain if you want to compile eLua for Cortex-M3 CPUs.

DISCLAIMER: I'm by no means a specialist in the GCC/newlib/binutils compilation process. I'm sure that there are better ways to accomplish what I'm describing here, however I just wanted a quick and dirty way to build a toolchain, I have no intention in becoming too intimate with the build process. If you think that what I did is wrong, innacurate, or simply outrageously ugly, feel free to contact me and I'll make the necessary corrections. And of course, this tutorial comes without any guarantees whatsoever.

Prerequisites

To build your toolchain you'll need:

- a computer running Linux: I use Ubuntu 8.04, but any Linux will do as long as you know how to find the equivalent of "apt-get" for your distribution. I won't be going into details about this, google it and you'll sure find what you need. It is also assumed that the Linux system already has a "basic" native toolchain installed (gcc/make and related). This is true for Ubuntu after installation. Again, you might need to check your specific distribution.
- GNU binutils: get it from here. At the moment of writing this, the latest versions is 2.18, which for some weird reason refuses to compile on my system, so I'm using 2.17 instead. **UPDATE**: you MUST use the new binutils 2.19 distribution for the Cortex toolchain, since it fixes some assembler issues. You won't be able to compile eLua 0.5 or higher if you don't use binutils 2.19.
- GCC: since support for Cortex (Thumb2) was only introduced staring with version 4.3.0, you'll need to download version 4.3.0 or newer. As I'm writing this, the latest GCC version is 4.3.1, which I'll be using for this tutorial. Download it from here after choosing a suitable mirror.
- Newlib: as I'm writing this, the latest official Newlib version is 1.16.0. However, the CVS version contains some fixes for the Thumb2 architecture, some of them in very important functions (like setjmp/longjmp), so you'll need to fetch the sources from CVS (this will most likely change when a new official Newlib version is released). So go to http://sourceware.org/newlib/download.html and follow the instructions there in order to get the latest sources from CVS.
- Also, the tutorial assumes that you're using bash as your shell. If you use something else, you might need to adjust some shell-specific commands.

Also, you need some support programs/libraries in order to compile the toolchain. To install them:

$ sudo apt-get install flex bison libgmp3-dev libmpfr-dev autoconf texinfo

Next, decide where you want to install your toolchain. They generally go in /usr/local/, so I'm going to assume /usr/local/cross-cortex for this tutorial. To save yourself some typing, set this path into a shell variable:

$ export TOOLPATH=/usr/local/cross-cortex

Step 1: binutils

This is the easiest step: unpack, configure, build.

$ tar xvfj binutils-2.19.tar.bz2
$ cd binutils-2.19
$ mkdir build
$ cd build
$ ../configure --target=arm-elf --prefix=$TOOLPATH --enable-interwork --enable-multilib --with-gnu-as --with-gnu-ld --disable-nls
$ make all
$ sudo make install
$ export PATH=${TOOLPATH}/bin:$PATH

Now you have your ARM "binutils" (assembler, linker, disassembler ...) in your PATH. They are fully capable of handling Thumb2.

Step 2: basic GCC

In this step we build a "basic" GCC (that is, a GCC without any support libs, which we'll use in order to build all the libraries for our target). But first we need to make a slight modification in the configuration files. Out of the box, the GCC 4.3.1/newlib combo won't compile properly, giving a very weird "Link tests are not allowed after GCC_NO_EXECUTABLES" error. After a bit of googling, I found the solution for this:

$ tar xvfj gcc-4.3.1.tar.bz2
$ cd gcc-4.3.1/libstdc++-v3
$ joe configure.ac

I'm using "joe" here as it's my favourite Linux text mode editor, you can use any other text editor. Now find the line which says "AC_LIBTOOL_DLOPEN" and comment it out by adding a "#" before it:

```
# AC_LIBTOOL_DLOPEN
```

Save the modified file and exit the text editor

$ autoconf
$ cd ..

Great, now we know it will compile, so let's do it:

$ mkdir build

$ cd build

$ ../configure --target=arm-elf --prefix=$TOOLPATH --enable-interwork --enable-multilib --enable-languages="c,c++" --with-newlib --without-headers --disable-shared --with-gnu-as --with-gnu-ld

$ make all-gcc

$ sudo make install-gcc

On my system, the last line above (sudo make install-gcc) terminated with errors, because it was unable to find our newly compiled binutils. If this happens for any kind of "make install" command, this is a quick way to solve it:

$ sudo -s -H

```
# export PATH=/usr/local/cross-cortex/bin:$PATH
# make install-gcc
# exit
```

Step 3: Newlib

Again, some modifications are in order before we start compiling. Because the CVS version of Newlib doesn't seem to have all the required support for Thumb2 yet, we need to tell Newlib to skip some of its libraries when compiling:

$ cd [directory where the newlib CVS is located]

$ joe configure.ac

Find this fragment of code:

```
arm-*-elf* | strongarm-*-elf* | xscale-*-elf* | arm*-*-eabi* )
 noconfigdirs="$noconfigdirs target-libffi target-qthreads"
 libgloss_dir=arm
 ;;
```

```
And add "target-libgloss" to the "noconfigdirs" variable:

   arm-*-elf* | strongarm-*-elf* | xscale-*-elf* | arm*-*-eabi* )
   noconfigdirs="$noconfigdirs target-libffi target-qthreads target-libgloss"
   libgloss_dir=arm
   ;;

Save the modified file and exit the text editor
$ autoconf
```

On one of the systems I ran the above sequence, it terminated with errors, complaining that autoconf 2.59 was not found. I don't know why that happens. 2.59 seems to be quite ancient, and the build ran equally well with 2.61 (the version of autoconf on the system that gave the error). If this happens to you, first execute autoconf --version to find the actual version of your autoconf, then do this:

```
$ joe config/override.m4
```

```
Look for this line:

  [m4_define([_GCC_AUTOCONF_VERSION], [2.59])])

And replace [2.59] with your actual version ([2.61] in my case).
$ autoconf
```

Once again, now we're ready to actually compile Newlib. But we need to tell it to compile for Thumb2. As already specified, I'm not a specialist when it comes to Newlib's build system, so I chosed the quick, dirty and not so elegant solution of providing the compilation flags directly from the command line. Also, as I wanted my library to be as small as possible (as opposed to as fast as possible) and I only wanted to keep what's needed from it in the final executable, I added the "-ffunction-sections -fdata-sections" flags to allow the linker to perform dead code stripping:

$ mkdir build

$ cd build

$ ../configure --target=arm-elf --prefix=$TOOLPATH --enable-interwork --disable-newlib-supplied-syscalls --with-gnu-ld --with-gnu-as --disable-shared

$ make CFLAGS_FOR_TARGET="-ffunction-sections -fdata-sections -DPREFER_SIZE_OVER_SPEED -D__OPTIMIZE_SIZE__ -Os -fomit-frame-pointer -mcpu=cortex-m3 -mthumb -D__thumb2__ -D__BUFSIZ__=256" CCASFLAGS="-mcpu=cortex-m3 -mthumb -D__thumb2__"

$ sudo make install

Some notes about the flags used in the above sequence:

- `--disable-newlib-supplied-syscalls:` this deserves a page of its own, but I won't cover it here. For an explanation, see for example this page.
- `-DPREFER_SIZE_OVER_SPEED -D__OPTIMIZE_SIZE__:` compile Newlib for size, not for speed (these are Newlib specific).
- `-mcpu=cortex-m3 -mthumb:` this tells GCC that you want to compile for Cortex. Note that you need both flags.
- `-D__thumb2__:` again, this is Newlib specific, and seems to be required when compiling Newlib for Cortex.
- `-Os -fomit-frame-pointer:` tell GCC to optimize for size, not for speed.
- `-D__BUFSIZ__=256:` again Newlib specific, this is the buffer size allocated by default for files opened via fopen(). The default is 1024, which I find too much for an eLua, so I'm using 256 here. Of course, you can change this value.

Step 4: full GCC

Finally, in the last step of our tutorial, we complete the GCC build. In this stage, a number of compiler support libraries are built (most notably libgcc.a). Fortunately this is simpler that the Newlib compilation step, as long as you remember that we want to build our compiler support libraries for the Cortex architecture:

$ cd gcc-4.3.1/build

$ make CFLAGS="-mcpu=cortex-m3 -mthumb" CXXFLAGS="-mcpu=cortex-m3 -mthumb

"LIBCXXFLAGS="-mcpu=cortex-m3 -mthumb" all
$ sudo make install

All Done!

Phew! That was quite a disturbing tutorial, with all that confusing flags lurking in every single shell line :) But at this point you should have a fully functional Cortex GCC toolchain, which seems to be something very rare, so enjoy it with pride. If you need further clarification, or if the above instructions didn't work for you, feel free to contact me.

Building GCC for i386

At first, the idea of an i386 "cross" compiler under Linux seems strange. After all, you're already running Linux on a i386 compatible architecture. But the compiler is sometimes tied in misterious ways with the operating system it's running on (see for example this page for some possible symptoms). And after all, you want to use Newlib, not libc, and to customize your development environment as much as possible. This tutorial will show you how to do that.

DISCLAIMER: I'm by no means a specialist in the GCC/newlib/binutils compilation process. I'm sure that there are better ways to accomplish what I'm describing here, however I just wanted a quick and dirty way to build a toolchain, I have no intention in becoming too intimate with the build process. If you think that what I did is wrong, innacurate, or simply outrageously ugly, feel free to contact me and I'll make the necessary corrections. And of course, this tutorial comes without any guarantees whatsoever.

› Prerequisites

To build your toolchain you'll need:

- a computer running Linux: I use Ubuntu 8.04, but any Linux will do as long as you know how to find the equivalent of "apt-get" for your distribution. I won't be going into details about this, google it and you'll sure find what you need. It is also assumed that the Linux system already has a "basic" native toolchain installed (gcc/make and related). This is true for Ubuntu after installation. Again, you might need to check your specific distribution.
- GNU binutils: get it from here. At the moment of writing this, the latest versions is 2.18, which for some weird reason refuses to compile on my system, so I'm using 2.17 instead.
- GCC: version 4.3.0 or newer is recommended. As I'm writing this, the latest GCC version is 4.3.1 which I'll be using for this tutorial. Download it from here after choosing a suitable mirror.
- Newlib: as I'm writing this, the latest official Newlib version is 1.16.0. Download it from the Newlib FTP directory.
- Also, the tutorial assumes that you're using bash as your shell. If you use something else, you might need to adjust some shell-specific commands.

Also, you need some support programs/libraries in order to compile the toolchain. To install them:

```
$ sudo apt-get install flex bison libgmp3-dev libmpfr-dev autoconf texinfo
```

Next, decide where you want to install your toolchain. They generally go in /usr/local/, so I'm going to assume /usr/local/cross-i686 for this tutorial. To save yourself some typing, set this path into a shell variable:

```
$ export TOOLPATH=/usr/local/cross-i686
```

› Step 1: binutils

This is the easiest step: unpack, configure, build.

$ tar xvfj binutils-2.17.tar.bz2
$ cd binutils-2.17
$ mkdir build
$ cd build
$ ../configure --target=i686-elf --prefix=$TOOLPATH --with-gnu-as --with-gnu-ld --disable-nls
$ make all
$ sudo make install
$ export PATH=${TOOLPATH}/bin:$PATH

Now you have your i386 "binutils" (assembler, linker, disassembler ...) in your PATH.

› Step 2: basic GCC

In this step we build a "basic" GCC (that is, a GCC without any support libs, which we'll use in order to build all the libraries for our target). But first we need to make a slight modification in the configuration files. Out of the box, the GCC 4.3.1/newlib combo won't compile properly, giving a very weird "Link tests are not allowed after GCC_NO_EXECUTABLES" error. After a bit of googling, I found the solution for this:

$ tar xvfj gcc-4.3.1.tar.bz2
$ cd gcc-4.3.1/libstdc++-v3
$ joe configure.ac

I'm using "joe" here as it's my favourite Linux text mode editor, you can use any other text editor. Now find the line which says "AC_LIBTOOL_DLOPEN" and comment it out by adding a "#" before it:

```
# AC_LIBTOOL_DLOPEN
```

Save the modified file and exit the text editor

$ autoconf
$ cd ..

Great, now we know it will compile, so let's do it:

$ mkdir build
$ cd build
$ ../configure --target=i686-elf --prefix=$TOOLPATH --enable-languages="c,c++" --with-newlib --without-headers --disable-shared --with-gnu-as --with-gnu-ld
$ make all-gcc
$ sudo make install-gcc

On my system, the last line above (sudo make install-gcc) terminated with errors, because it was unable to find our newly compiled binutils. If this happens for any kind of "make install" command, this is a quick way to solve it:

$ sudo -s -H
```
# export PATH=/usr/local/cross-i686/bin:$PATH
# make install-gcc
# exit
```

› Step 3: Newlib

Once again, Newlib is as easy as unpack, configure, build. But I wanted my library to be as small as possible (as opposed to as fast as possible) and I only wanted to keep what's needed from it in the final executable, so I added the "-ffunction-sections -fdata-sections" flags to allow the linker to perform dead code stripping:

$ tar xvfz newlib-1.16.0.tar.gz
$ cd newlib-1.16.0
$ mkdir build
$ cd build
$ ../configure --target=i686-elf --prefix=$TOOLPATH --disable-newlib-supplied-syscalls --with-gnu-ld --with-gnu-as --disable-shared
$ make CFLAGS_FOR_TARGET="-ffunction-sections -fdata-sections-DPREFER_SIZE_OVER_SPEED -D__OPTIMIZE_SIZE__ -Os -fomit-frame-pointer -D__BUFSIZ__=256"
$ sudo make install

Some notes about the flags used in the above sequence:

- `--disable-newlib-supplied-syscalls:` this deserves a page of its own, but I won't cover it here. For an explanation, see for example this page.
- `-DPREFER_SIZE_OVER_SPEED -D__OPTIMIZE_SIZE__:` compile Newlib for size, not for speed (these are Newlib specific).
- `-Os -fomit-frame-pointer:` tell GCC to optimize for size, not for speed.
- `-D__BUFSIZ__=256:` again Newlib specific, this is the buffer size allocated by default for files opened via fopen(). The default is 1024, which I find too much for an eLua, so I'm using 256 here. Of course, you can change this value.

› Step 4: full GCC

Finally, in the last step of our tutorial, we complete the GCC build. In this stage, a number of compiler support libraries are built (most notably libgcc.a). Fortunately this is simpler that the Newlib compilation step:

$ cd gcc-4.3.1/build
$ make all
$ sudo make install

› Step 5: all done!

Now you can finally enjoy your i386 toolchain, and compile eLua with it :) After you do, you'll be able to boot eLua directly on your PC, as described here, but you won't need to download the ELF file from the eLua project page, since you just generated it using your own toolchain! If you need further clarification, or if the above instructions didn't work for you, feel free to contact me.

Booting your PC in eLua

That's right: after following this tutorial, your PC will boot directly into Lua! No OS there (this explains why the boot process is so fast), just you and Lua. You'll be able to use the regular Lua interpreter to write your programs and even use "dofile" to execute Lua code.

Details

Booting Lua involves using the well known GRUB that will be used to load a multiboot compliant ELF file that contains our eLua code. Since the eLua code and the build instructions are not available yet, I'll be providing a direct link to the ELF file. The code runs in protected mode, so you have access to your whole memory. The code does not access any kind of storage device (HDD, CDROM, floppy), so if you're worried that it might brick your system, you can relax now :) I'm only using some very basic keyboard and VGA "drivers", so all you're risking is a system freeze (even this is highly unlikely), nothing a good old RESET can't handle (be sure to use the hardware reset though, CTRL+ALT+DEL is not handled by the code). But just in case, see also the next section.

Disclaimer

As already mentioned, the code won't try to access any kind of storage (HDD, CDROM, floppy), not even for reading, so you don't need to worry about that. Also it doesn't try to reprogram your video card registers, so it can't harm it or your monitor. It only implements a "protected mode keyboard driver" that can't physically damage anything in your system. In short, I made every effort to make the code as harmless as possible. I tested it on 5 different computers and in 2 VirtualBox emulators, and nothing bad happened. That said, there are no warranties of any kind. In the very unlikely event that something bad does happen to your system, you have my sincere sympathy, but I can't be held responsible for that.

Prerequisites

To boot your computer in Lua you'll need:

- a 386 or better computer running Linux. I actually tested this only on Pentium class computers, but it should run on a 386 without problems.
- GRUB. Since you're running Linux, chances are you're already using GRUB as your bootloader. If not, you must install it. You don't need to install it on your HDD; a floppy, an USB stick or even a CDROM will work as well. I won't cover the GRUB installation procedure here, just google for "install grub on floppy/usb/cdrom" and you'll sure find what you're looking for. You can try for example here, here or here.
- The eLua ELF file. Download it from here. OR download eLua and compile it for the i386 architecture using a toolchain that you can build by following this tutorial.
- a text editor to edit your GRUB configuration file.

The rest of this tutorial assumes that you're using Linux with GRUB, and that GRUB is located in /boot/grub, which is true for many Linux distributions (I'm using Ubuntu 8.04).

Let's do this

First, copy the eLua ELF file to your "/boot" directory:

```
$ sudo cp surprise /boot
```

Next you need to add another entry to your GRUB menu file (/boot/grub/menu.lst). Edit it and add this entry:

```
title Surprise!
root (hd0,0)
kernel /boot/surprise
boot
```

You may need to modify the root (hd0,0) line above to match your boot device. The best way to do this is to look in the menu.lst file for the entry that boots your Linux kernel. It should look similar to this:

```
title           Ubuntu, kernel 2.6.20-16-generic
root            (hd0,2)
kernel          /boot/vmlinuz-2.6.20-16-generic
initrd          /boot/initrd.img-2.6.20-16-generic
savedefault
```

After you find it, simply use the root (hdx,y) line from that entry (root (hd0,2) in the example above) in your newly created entry instead of root (hd0,0). That's it! Now reboot your computer, and when the GRUB boot menu appears, choose "Surprise!" from it. You can even type dofile "/rom/bisect.lua" to execute the "bisect.lua" test file. Enjoy! As usual, if you need more details, you can contact us. Also, if you want to have you own USB stick that boots Lua, let me know. If enough people manifest their interest in this, I'll add another tutorial on how to do it (I already have an USB stick that boots Lua, of course :)).

Booting eLua from a stick

This is follow up of this tutorial. After completing it you'll be able to boot eLua directly from your USB stick (provided, of course, that your computer can boot from an USB stick, which is true for most computers nowadays). You might want to check the boot your PC in eLua tutorial first for more details. If you have an old USB stick that you don't use anymore, and/or the shear geekness of this idea makes you feel curious, this tutorial is definetely for you :)

Disclaimer

As mentioned here, the code won't try to access any kind of storage (HDD, CDROM, floppy), not even for reading, so you don't need to worry about that. Also it doesn't try to reprogram your video card registers, so it can't harm it or your monitor. It only implements a "protected mode keyboard driver" that can't physically damage anything in your system. In short, I made every effort to make the code as harmless as possible. I tested it on 5 different computers and in 2 VirtualBox emulators, and nothing bad happened. That said, there are no warranties of any kind. In the very unlikely event that something bad does happen to your system, you have my sincere sympathy, but I can't be held responsible for that. Also, I can't be held responsible if you mess up your HDD by failing the GRUB installation procedure (even though, once again, this shouldn't be possible unless you really insist on messing it up). If you're new to computers, this tutorial might not be for you. Your call.

Prerequisites

To have your own bootable eLua USB stick you'll need:

- an USB stick. I tested this on an 128M USB stick, because it's the smallest I could find. You should be OK with a 4M stick or even a 2M stick
- a computer running Linux. I use Ubuntu, but any other distribution is fine.
- GRUB. Since you're running Linux, chances are you're already using GRUB as your bootloader. If not, you must install it on your HDD, or at least know how to install it directly on the USB stick. I won't go into details here, google it and you'll find lots of good articles about GRUB. This tutorial assumes that you're using GRUB as your bootloader.
- The eLua ELF file. Download it from here. OR download eLua and compile it for the i386 architecture using a toolchain that you can build by following this tutorial.
- a text editor to edit your GRUB configuration file.

The rest of this tutorial assumes that you're using Linux with GRUB, and that GRUB is located in /boot/grub, which is true for many Linux distributions.

Backup your stick

Since the stick is going to be formatted, make sure to backup the data from your stick first (you can copy it back after finishing the tutorial).

Partition and format your stick

Depending on your stick, this step might not be required, but chances are you'll need to re-partition and re-format your stick before installing GRUB on it. The problem is that many sticks have a very creative, non-standard partition table, and GRUB doesn't like that. I looked at the partition table on my eLua USB stick, and it scared me to death, so I had to follow this procedure. In short, you'll need to delete all the partitions from your stick, create a new partition, and then format it. For a step by step tutorial check here.

Install GRUB on your stick

First, mount your freshly formatted stick (I'm going to assume that the mount directory is /mnt):

```
$ sudo mount /dev/sda1 /mnt
```

(of course, you'll need to change /dev/sda1 to reflect the physical location of your USB stick). Then copy the required GRUB files to your stick:

```
$ cd /mnt
$ mkdir boot
$ mkdir boot/grub
$ cd /boot/grub
$ cp stage1 fat_stage1_5 stage2 /mnt/boot/grub
```

Copy the eLua ELF file to the GRUB directory as well:

```
$ cp surprise /mnt/boot/grub
```

Create a menu.lst file for GRUB with you favourite text editor (I'm using joe):

```
$ cd /mnt/boot/grub
$ joe menu.lst
 title Surprise!
 root (hd0,0)
 kernel /boot/grub/surprise
 boot
```

Now it's time to actually install GRUB on the stick.

```
  $ sudo -s -H
  # grub
  Now we need to find the GRUB name of our USB stick. We'll use the "find" command
from
  GRUB and our "surprise" file to accomplish this:

  grub> find /boot/grub/surprise
  (hd2,0)

  GRUB should respond with a single line (like (hd2,0) above). If it gives you
more
  than one line, something is wrong. Maybe you also installed eLua on your HDD? If
so,
  delete the /boot/grub/surprise file from your HDD and try again.
  You might get a different (hdx,y) line. If so, just use it instead of (hd2,0) in
the rest of
  this tutorial.

  grub> root (hd2,0)
  grub> setup (hd2)
```

```
 Checking if "/boot/grub/stage1" exists... yes
 Checking if "/boot/grub/stage2" exists... yes
 Checking if "/boot/grub/fat_stage1_5" exists... yes
 Running "embed /boot/grub/fat_stage1_5 (hd2)"...  15 sectors are embedded.
 succeeded
 Running "install /boot/grub/stage1 (hd2) (hd2)1+15 p (hd2,0)/boot/grub/stage2
 /boot/grub/menu.lst"... succeeded
Done.
grub> quit
```

That's it! Now reboot your computer, make sure that your BIOS is set to boot from USB, and enjoy! You can even type dofile "/rom/bisect.lua" to execute the "bisect.lua" test file. As usual, if you need more details, you can contact us.

Using OpenOCD

Quick downloads

If you'd rather skip the long and boring OpenOCD introduction and skip directly to the OpenOCD script downloads, use the linke below.

`Configuration files for STR9-comStick`

`Configuration files for LPC2888`

`Configuration files for STR7`

About OpenOCD

OpenOCD is an open source tool that can be used to connect to a CPU's JTAG interface. Using OpenOCD and a physical JTAG connection allows you to burn the on-chip flash memory of your CPU (or to load your code directly to RAM), to read the internal CPU memory (Flash/RAM) and to use gdb to debug your code. Needless to say, this is a very handy tool (and especially handy if your CPU happens to be built around an ARM core, since in this case you can be almost certain that it has a JTAG interface that you can use). That said, if your only goal is to burn your firmware, my personal suggestion is to avoid using OpenOCD if possible. It has quite a steep learning curve, because it is a command line tool that uses configuration files with lots of different parameters, and this takes a while to get used to. Worse, I feel that it is not very well docummented. The project's wiki does give a few good pointers about all the configuration parameters, and there are some good OpenOCD tutorials out there, but none of them tells the whole story. And the syntax (and even some commands) seems to change slightly between releases, which makes things even more confusing. This is why I generally choose to use a different firmware burning tool when available, and resort to OpenOCD only for targets that lack a proper firmware burning tool. If you need to debug your code, however, you probably want to use OpenOCD, since the alternatives aren't cheap. To summarize, you can forget about OpenOCD when:

- your CPU manufacturer provides a special tool for firmware burning. This is quite often the case, but more often that not the forementioned tools work only in Windows.
- you must debug your code, but you have a good intuition about where the problem is located. In this case, simply connecting a LED to a PIO port and turning it on and off from different parts of your code until you figure out exactly what's the problem can work wonders. I can't remember when was the last time I used gdb for debugging, since "LED debugging" was all I needed.

On the other hand, you should probably use OpenOCD when:

- your CPU manufacturer doesn't provide a special tool for firmware burning (or it does, but it's not what you need).
- you're using Linux, MacOS or another OS that is not supported by the firmware burning tool.
- you need to do some serious debugging in order to understand what's wrong with your application.

If you decided that you don't need OpenOCD after all, now it's a good time to navigate away from this page and save yourself from some possible symptoms of headache. If you need OpenOCD, read on, I'll try to make this as painless as possible. However, don't expect this to be a full tutorial on OpenOCD, because it's not; my intention is to give you just enough data to use OpenOCD for burning eLua on your board. Because of this, I won't be covering debugging with OpenOCD here, just firmware burning. And, before we begin, please read and understand the next paragraph.

DISCLAIMER: using OpenOCD improperly may force your CPU to behave unexpectedly. While physically damaging your CPU as a result of using OpenOCD is very hard to accomplish, you might end up with a locked chip, or you might erase a memory area that was not supposed to be erased, you might even disable the JTAG interface on your chip (thus rendering it unusable). If you modify the configuration scripts that I'm going to provide, make sure that you know what you're doing. Also, I'm not at all an OpenOCD expert, so my configuration scripts might have errors, even though I tested them. In short, this tutorial comes without any guarantees whatsoever.

Getting OpenOCD

If you're on Windows, the best place to get OpenOCD already compiled and ready to run is to visit the Yagarto home page. They provide a very nice OpenOCD installer, and they seem to keep up with OpenOCD progress (the versions on the Yagarto site are not "bleeding edge", but there are quite fresh nevertheless). If you're on Linux, you can always use apt-get or your distribution-specific package manager:

```
$ sudo apt-get install openocd
```

There is a catch here though: the OpenOCD version that I get from apt-get is dated 2007-09-05, while the Yagarto OpenOCD version is from 2008-06-19. Since I'm using OpenOCD from Windows (because Ubuntu 8.04 doesn't seem to handle my USB-to-JTAG adapters very well), my instructions are relevant to the Yagarto version. As mentioned in the introduction, the meaning and parameters of different commands might change between OpenOCD version, so if you want to use the Yagarto version on your non Windows system, you'll have to build it from source (see below). The main resource on how to build OpenOCD from source is the OpenOCD build page from the OpenOCD wiki. Also, a very good tutorial can be found here. I'm not going to provide step by step build instructions, since the two links that I mentioned cover this very well, and the build process is relatively straightforward. However, since both tutorials describe how to build the bleeding edge version of OpenOCD, you'll need a slight modification do build the Yagarto version instead. The modification is in the SVN checkout step. Replace this step:

```
$ svn checkout svn://svn.berlios.de/openocd/trunk
```

With this step ('717' is the SVN revision of the Yagarto OpenOCD build):

```
$ svn checkout -r 717 svn://svn.berlios.de/openocd/trunk
```

Follow the rest of the build instructions, and in the end you should have a working OpenOCD.

Supported targets

I couldn't find a good page with a list of the targets that are supported by OpenOCD. So, if you want to check if your particular CPU is supported by OpenOCD, I recommend getting the latest sources (as described in the previous section) and listing the trunk/src/target/target directory:

```
$ ls trunk/src/target/target
```

```
at91eb40a.cfg
at91r40008.cfg
cfi.c
....
str9comstick.cfg
....
```

If this listing has something that looks like your CPU name, you're in luck. OpenOCD has support for LPC from NXP, AT91SAM cfrom Atmel, STR7/STR9 from ST, and many others.

Using OpenOCD

To use OpenOCD, you'll need:

- the OpenOCD executable, as described above
- a board with a JTAG interface
- a JTAG adapter

In some cases, your CPU board might provide a built in JTAG adapter. For example, my LM3S8962 board provides both an USB-to-JTAG and an USB-to-serial converter built on board, switching between them automatically. The same is true for my STR9-comStick. On the other hand, my SAM7-EX256 board has only a JTAG connector, I need a separate JTAG adapter to connect to it. I'm using ARM-USB-TINY from Olimex, but there are other affordable USB-to-JTAG adapters out there, like the Amontec JTAGKey-Tiny. Not to mention that you can build your ownt. Although USB is my interface of choice, you'll find JTAG adapters for PC LPT ports too. The good news is that once you buy a JTAG adapter, chances are that it will work with many boards with different CPUs, since the JTAG connector layout is standardized and the JTAG adapters are generally able to work with different voltages. To actually use OpenOCD, you'll need a configuration file. The configuration file is the one that lets OpenOCD know about your setup, such as:

```
* the kind of JTAG interface that you're using.
* the actual hardware platform you're using (ATM7TDMI, ARM966 and others).
* the memory configuration of your CPU (flash banks).
* the script used to program the flash memory.
```

Presenting a list of all the possible configuration options and their meaning is way beside the scope of this document, so I'm not going to do it, I'll give an example instead. For the example I'm going to use parts of my STR-comStick configuration file (comstick.cfg) adapted from the OpenOCD distribution and from other examples (don't worry, I'll provide full download links for this file later on). First we need to tell OpenOCD that we're using a the STR9-comStick USB-to-JTAG adapter:

```
interface ft2232
ft2232_device_desc "STR9-comStick A"
ft2232_layout comstick
ft2232_vid_pid 0x0640 0x002C
jtag_speed 4
jtag_nsrst_delay 100
jtag_ntrst_delay 100
```

Also, OpenOCD needs to know what's our target and its memory layout:

```
target arm966e little run_and_init 1 arm966e
run_and_halt_time 0 50
```

```
working_area 0 0x50000000 32768 nobackup

flash bank str9x 0x00000000 0x00080000 0 0 0
flash bank str9x 0x00080000 0x00008000 0 0 0
```

This tells OpenOCD that our target is an ARM966-E running in little endian mode, with two flash memory banks, one that starts at 0x0 and it's 0x80000 bytes in size, and another one that starts at 0x80000 and it's 0x8000 bytes in size. Finally, OpenOCD must know what's the name of our script file (this is the file that is used to pysically program the CPU memory):

```
#Script used for FLASH programming
target_script 0 reset str91x_flashprogram.script
```

The contents of the str91x_flashprogram.script is very target-dependent:

```
wait_halt
str9x flash_config 0 4 2 0 0x80000
flash protect 0 0 7 off
flash erase_sector 0 0 7
flash write_bank 0 main.bin 0
reset run
sleep 10
shutdown
```

I'm not even going to attempt to explain this one :) Basically it unprotects the flash, erases it, writes the contents of "main.bin" to flash, and then resets the CPU. If you need to flash a file with a different name, the only thing you need to modify is the "main.bin" in the "flash write_bank" line. To use all this, you need to tell OpenOCD to use our configuration file:

```
openocd-ftd2xx -f comstick.cfg
```

(note: under Windows, the OpenOCD executable name is often "openocd-ftd2xx". Under Linux it's simply "openocd". Replace it with the actualy name with your executable.) That's it for your OpenOCD crash course. I realise that there's much more to learn, so here's a list of links with much better information on the subject:

- OpenOCD quick reference card. (slightly outdated)
- A very good OpenOCD tutorial.
- OpenOCD configuration examples from the official OpenOCD wiki.
- An excellent page about using OpenOCD with ARM controllers, with lots of real life examples.
- An interesting topic on the SparkFun forum about STR9 and OpenOCD.

Configuration files for STR9-comStick

Download them below:

comstick.cfg

str91x_flashprogram.script

comrst.cfg

str91x_reset.script

The comstick.cfg configuration file is for prorgramming the STR9-comStick. comrst.cfg is for resetting it. The comStick has a very interesting habit: after you power it (via USB) it does not start executing the code from the internal flash, you need to execute OpenOCD with the comreset.cfg script to start it. This script does exactly what it says: executes a CPU reset (since the board doesn't have a RESET button). This is a very peculiar behaviour, and I'm not sure if it's generic or it's only relevant to my particular comStick. I suspect that the CPU RESET line isn't properly handled by the on-board USB-to-JTAG converter, and the only solution I have for this is to execute this script everytime you power the board and everytime you need to do a RESET.

Configuration files for LPC2888

LPC2888 is quite a different animal. I couldn't find any "official" LPC2888 configuration file for OpenOCD, so I had to learn how to write my own. It works, but I suspect it can be improved. This time, the configuration file applies to the latest (SVN) version of OpenOCD, so read this tutorial to understand how to get the latest OpenOCD sources and how to compile them (this section is based on version 922 of the OpenOCD repository). Then use the next file to burn your binary image to the chip:

lpc2888.cfg

If your image name is not main.bin edit the file and change the corresponding line (flash write_bank 0 main.bin 0), then invoke openocd like this:

```
openocd -f lpc2888.cfg
```

I'm using ARM-USB-TINY from Olimex, but it should be easy to use the script with any other JTAG adapter (don't forget to change the script to match your adapter).

Configuration files for STR711FR2 (STR7 from ST)

Download them below:

str7prg.cfg

str7_flashprogram.script

str7rst.cfg

str7_reset.scrip

For STR7 I'm using the Yagarto OpenOCD build for Windows (repository version 717, as described at the beginning of this tutorial). The str7prg.cfg configuration file is for prorgramming the STR9-comStick. str7rst.cfg is for resetting it. I'm using a STR711FR2 heard board from ScTec to which I attached a few LEDs and a MAX3232 TTL to RS232 converter for the serial communication. The board comes with its own JTAG adadpter, but it uses a parallel interface, and since my computer doesn't have one, I used the ARM-USB-TINY from Olimex. To use them, invoke the OpenOCD executable like this:

```
openocd-ftd2xx -f str7prg.cfg
```

(note: under Windows, the OpenOCD executable name is often "openocd-ftd2xx". Under Linux it's simply "openocd". Replace it with the actualy name with your executable.) Also, be sure to modify str7_flashprogram.script if your image name is not main.bin.

Apendice A

Previous documentation files

eLua components
===

Besides the Lua core, the platform modules
(docs/platform_modules.txt) and the
Newlib 'glue code', eLua uses a number of other code modules
(components) for
extended functionality. This is a great thing if you actually need
the whole
functionality in your code, but otherwise it becomes a waste of
memory space.
Since eLua was designed to be as flexible as possible, it includes a
mechanism
that allows the user to select exactly what components he needs. Only
the
selected components will be part of the eLua image. Please note that
this is
not a replacement of the platform modules mechanism, the two are
complementary
to each other, since the components are not connected with the
platform
interface (docs/platform_interface.txt) in any way.
To use this feature, every platform (src/platform/[name]) must
include a file
named "platform_conf.h" that specifies (among other things) what
components
should be built for that platform. For example, the LM3S
"platform_conf.h" file
might look like this:

```
(BEGIN src/platform/lm3s/platform_conf.h)
// Define here what components you want for this platform

#ifndef __PLATFORM_CONF_H__
#define __PLATFORM_CONF_H__

#define BUILD_XMODEM
#define BUILD_SHELL
#define BUILD_ROMFS
#define BUILD_TERM
........................................
(END src/platform/lm3s/platform_conf.h)
```

In this case, the XMODEM, SHELL, ROMFS and TERM components will be
built. On the
other hand, the i386 "platform_conf.h" will probably have less
components:

```
(BEGIN src/platform/i386/platform_conf.h)
// Define here what components you want for this platform

#ifndef __PLATFORM_CONF_H__
#define __PLATFORM_CONF_H__

#define BUILD_ROMFS
#define BUILD_SHELL
........................................
(END src/platform/i386/platform_conf.h)
```

You don't need to modify any other part of your code, just rebuild your image
after you made changes to this file (docs/building.txt)
Below you can find a list of eLua components and their functionality.

XMODEM
==
The XMODEM component enables eLua to receive Lua source files via its shell and
execute them (docs/the_elua_shell.txt). If you don't need to use "recv" from the
shell you can skip this component.
To enable:

```
  #define BUILD_XMODEM
```

Also, XMODEM is configured with a number of constants also defined in platform_conf.h. They are:

XMODEM_UART_ID : the id of the UART on which XMODEM runs
XMODEM_TIMER_ID : the id of the timer used by the XMODEM implementation

SHELL

```
==========================================================================
This enables the build of the eLua shell (docs/the_elua_shell.txt).
If you don't
need the shell, don't enable this component. eLua will execute the
"lua" command
at startup if the eLua shell is not built. The shell comes in two
flavours: over
a serial line or over TCP/IP (currently you can't have both at the
same time).
To enable shell over a serial line:
  #define BUILD_SHELL
  #define BUILD_CON_GENERIC
To enable shell over TCP/IP:
  #define BUILD_SHELL
  #define BUILD_CON_TCP
```

ROMFS

==

If you need to use the ROM file system (docs/the_rom_file_system.txt) enable
this component, otherwise you can skip it.
To enable:

```
  #define BUILD_ROMFS
```

TERM

==
The TERM module adds support for ANSI terminals. See
docs/terminal_support.txt
for details. If you don't need it, and if you're willing to miss the
opportunity
of playing hangman in eLua (examples/hangman.lua) you can skip this
component :)
To enable:

```
  #define BUILD_TERM
```

Also, TERM is configured with a number of constants also defined in
platform_conf.h. THey are:

TERM_UART_ID - the id of the UART on which TERM runs
TERM_TIMER_ID - the id of the timer used by the TERM implementation
TERM_LINES - number of lines in the terminal emulator
TERM_COLS - number of columns in the terminal emulator
TERM_TIMEOUT - inter-key timeout (used to detect keys that send
multiple codes,
  such as up/down/left/right keys).

IMPORTANT NOTE: TERM doesn't currently work over TCP/IP.

uIP
===
uIP is the TCP/IP stack used curently by eLua to provide networking support
(docs/tcpip_in_elua.txt). You can enable the TCP/IP stack and two of its services
(the DHCP client and the DNS resolver).
To enable uIP (thus TCP/IP support):

```
  #define BUILD_UIP
```

To enable the DHCP client:

```
  #define BUILD_DHCPC
```

To enable the DNS resolver:

```
  #define BUILD_DNS
```

eLua generic modules
==

Before you read this file, please make sure that you read and understood the
theory from "platform_modules.txt" (at least the first part which describes
the platform modules implementation). The generic modules use the exact same
mechanism. In fact, the only difference between them is that the generic modules
are exactly what their name implies: generic. They don't depend on the platform
interface, so they don't need specific support for each platform, but they still
behave identically on all platforms. They are selected by the same mechanism
used by the platform modules (platform_libs.h). Following is a list of these
modules and their exported functions.

The "term" module
==
The "term" component (see terminal_support.txt) exports its functions to Lua via
the "term" module. The methods of the "term" module are presented below.

term.clrscr(): clear the screen

term.clreol(): clear from the current cursor position to the end of the line

term.gotoxy(x, y): position the cursor at the given coordinates

term.up(delta): move the cursor up "delta" lines

term.down(delta): move the cursor down "delta" lines

term.left(delta): move the cursor left "delta" lines

term.right(delta): move the cursor right "delta" lines

Lines = term.lines(): returns the number of lines

Cols = term.cols(): returns the number of columns

term.put(c1, c2, ...): writes the specified character(s) to the terminal

term.putstr(s1, s2, ...): writes the specified string(s) to the terminal

Cx = term.cursorx(): return the cursor X position

Cy = term.cursory(): return the cursor Y position

c = term.getch(term.WAIT | term.NOWAIT): returns a char read from the
  terminal. If term.WAIT is specified the function will wait for a character to
  be ready, with term.NOWAIT it returns -1 if no char is available or the char
  code if a char is available. The return of getch can be checked against the
  char codes defined in inc/term.h, by appending "term." to the constant name
  (for example: term.KC_UP, term.KC_LEFT, term.KC_ESC ...)

The "pack" module

Pack allows packing/unpacking of binary data. For example, it allows one to save a specific data type (for example a 16-bit integer) from Lua to a file and then read it back to a Lua variable without having to worry about the different physical representations of a Lua number and a 16-bit integer. It's originally written by Luiz Henrique de Figueiredo, one of the "fathers" of Lua. It exports just two methods ("pack" and "unpack"), but it uses some format specifiers for the pack/unpack operations that take a while to get used to. For more information download the original distribution and check its documentation and examples (http://www.tecgraf.pub-rio.br/~lhf/ftp/lua/~lpack).

The "bit" module

As Lua doesn't have built-in operators for bit operations (and, or, xor, not) they are provided by this module. It's based on "bitlib" by Reuben Thomas and was slightly adapted and augmented for eLua.

Res = bit.bnot(value): unary negation

Res = bit.band(v1, v2, ...): binary "and"

Res = bit.bor(v1, v2, ...): binary "or"

Res = bit.bxor(v1, v2, ...): binary "exclusive or"

Res = bit.lshift(value, pos): shift "value" left "pos" positions.

Res = bit.rshift(value, pos): shift "value" right "pos" positions. The sign is
  not propagated.

Res = bit.arshift(value, pos): shift "value" right "pos" positions. The sign
  is propagated ("arithmetic shift").

Res = bit.bit(bitno): a shortcut for bit.lshift(1, bitno)

Res1, Res2, ... = bit.set(bitno, v1, v2, ...): set the bit at position "bitno"
  in v1, v2, ... to 1.

Res1, Res2, ... = bit.clear(bitno, v1, v2, ...): set the bit at position
  "bitno"in v1, v2, ... to 0.

Res = bit.isset(value, bitno): returns true if bit at position "bitno" in
  "value" is 1, false otherwise.

Res = bit.isclear(value, bitno): returns true if bit at position "bitno" in
  "value" is 0, false otherwise.

The "math" module

This is actually part of the official Lua distribution, not a separate module.
Its purpose is to provide mathematic functiosn (sin, cos, tan...) to Lua. Since
these kind of functions are rarely needed in an embedded environment, the "math"
module can be enabled and disabled just like the other generic and platform
modules in eLua.

The "net" module
==
TCP/IP networking support is provided to eLua via the "net" module. It contains a small set of function, tailored for embedded systems (lighter and less resource demanding).

IMPORTANT NOTE: TCP/IP support in eLua is still largely experimental.

Sock = socket(type): creates a new socket and returns its identifier. type can be either net.SOCK_STREAM or net.SOCK_DGRAM, but currently only TCP/IP sockets (SOCK_STREAM) are implemented.

Res = close(socket): closes the given socket, returning an error status.

IP = packip(ip0, ip1, ip2, ip3) or IP = packip("ipstring"): packs the given IP (either in unpacked form or as a string) returning a value that completely identifies that IP (it's not actually a new IP datatype, just a 32-bit number).

IP0, IP1, IP2, IP3 = unpackip(ip, "*n") or IPString = unpackip(ip, "*s"): unpacks the given IP value, returning it either as 4 numbers or as a string.

Sock, RemoteIp, Err = accept(port, [timer_id, timeout]): listens on the specified port, waiting for connections. If timer ID and timeout are specified, it uses the specified timer to wait for a connection for at most "timeout" microseconds. Returns the socket descriptor for the new connection, the IP of the remote end, and an error status.

Sent, Res = send(sock, string): send the given "string" on the specified socket, returning the number of bytes actually send and an error status.

Data, Res = recv(sock, maxsize, [timer_id, timeout]) or
Data, Res = recv(sock, "*l", [timer_id, timeout]): read data from

the
socket. The first form reads up to a maximum size specified by "maxsize", the
second form reads a single line (until '\n' is received, ignoring any '\r'
chars in the stream).
IMPORTANT NOTE: currently, the "*l" (line) mode is partially broken, in that it
might loose some of the data sent by the remote end. If the remote end sends
more than one line, only the first is kept, the rest is ignored. For example,
if the remote sends "line\n", everything is OK, but if the remote sends
"line1\nline2\n", "line1" is returned correctly after calling "recv" once, but
"line2" won't be returned after calling "recv" again. This is due to the "single
buffer" design of uIP. If you want to make sure you receive all the data you're
looking for, use the first form of recv, specifying a maximum size. "*l" is only
usable for line-oriented conversations (like you'd find in a command line shell,
for example).

IP = lookup("hostname"): invokes the DNS resolver to find the IP address of
"hostname".

Err = connect(sock, ip, port): connects the specified socket (that must be
created previously using "socket") to the specified host and port. Returns an
error status.

The error status is defined in inc/elua_net.h in one enum:

```
(BEGIN inc/elua_net.h)
// eLua net error codes
enum
{
  ELUA_NET_ERR_OK = 0,
  ELUA_NET_ERR_TIMEDOUT,
  ELUA_NET_ERR_CLOSED,
  ELUA_NET_ERR_ABORTED,
  ELUA_NET_ERR_OVERFLOW
};
```

```
(END inc/elua_net.h)
```

The "disp" module

The disp module provides support for graphical displays.
This documentation is somewhat focused on the single display supported so far.

Platforms supported: LM3S8962
Displays supported: RIT Oled display RIT128x96x4

Module functions:

disp.init(freq)
This function initializes the SSI interface to the OLED display and

configures the SSD1329 controller on the panel.

Parameters:
freq: specifies the SSI Clock Frequency to be used.

Obs:
The LM3S8962 Luminary Micro board can be initialized with disp.init(100000)

disp.enable(freq)
This function initializes the SSI interface to the OLED display

Parameters:
freq: specifies the SSI Clock Frequency to be used.

Obs:
The LM3S8962 Luminary Micro board can be enabled with disp.enable(100000)

disp.disable()
This function frees the SSI interface to be used to some other function.

Parameters:
None.

disp.on()
This function will turn on the OLED display, causing it to display the
contents of its internal frame buffer.

Parameters:
  None.

disp.off()
  This function will turn off the OLED display. This will stop the scanning
 of the panel and turn off the on-chip DC-DC converter, preventing damage to
 the panel due to burn-in (it has similar characters to a CRT in this
 respect).

Parameters:
  None.

disp.stringdraw(str, x, y, lvl)
  This function prints a string on the an x, y pixel position on the graphical display.

Parameters:
  str: string to be printed
  x: screen column position in pixels (0-127)
  y: screen line position in pixels (0-95)
  lvl: intensity level / gray level (0-15)

Obs:
  Only the ASCII characters
 between 32 (space) and 126 (tilde) are supported. Other characters will
 result in random data being draw on the display.
  If the drawing of the string reaches the right edge of the display, no more
 characters will be drawn. Therefore, special care is not required to avoid
 supplying a string that is too long to display.
  Because the OLED display packs 2 pixels of data in a single byte, the
 parameter x must be an even column number.

disp.imagedraw(img, x, y, w, h)
  This function will display a bitmap graphic on the display.

```
Parameters:
  img: image stream to be printed
  x: screen column position in pixels (0-127)
  y: screen line position in pixels (0-95)
  w, h: image width and height dimensions

Obs:
  Because of the
 format of the display RAM, the starting column (x) and the number of
 columns (w) must be an integer multiple of two.

  The image data is organized with the first row of image data
appearing left
 to right, followed immediately by the second row of image data.
Each byte
 contains the data for two columns in the current row, with the
leftmost
 column being contained in bits 7:4 and the rightmost column being
contained
 in bits 3:0.

  For example, an image six columns wide and seven scan lines tall
would
 be arranged as follows (showing how the twenty one bytes of the
image would
 appear on the display):

+-------------------+-------------------+-------------------+

|      Byte 0       |      Byte 1       |      Byte 2       |

+---------+---------+---------+---------+---------+---------+

| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |

+---------+---------+---------+---------+---------+---------+

|      Byte 3       |      Byte 4       |      Byte 5       |

+---------+---------+---------+---------+---------+---------+

| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |

+---------+---------+---------+---------+---------+---------+
|      Byte 6       |      Byte 7       |      Byte 8       |

+---------+---------+---------+---------+---------+---------+
```

```
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 9       |      Byte 10      |      Byte 11      |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 12      |      Byte 13      |      Byte 14      |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
```

Console input/output in eLua
==

NOTE: this document describes the terminal support over serial connections
only. Refer to docs/elua_components.txt to learn how to enable console
support over TCP/IP instead of serial connections.

The console input/output is handled by a generic layer (src/newlib/genstd.c)
that can be adapted to a variety of input/output devices. It needs just two
functions, one for displaying characters and another one for receiving input:

```
(BEGIN inc/newlib/genstd.h)
// Send/receive function types
typedef void ( *p_std_send_char )( int fd, char c );
typedef int ( *p_std_get_char )();
(END inc/newlib/genstd.h)
```

(the send faction gets an additional 'fd' parameter that you can use to
differentiate between STDOUT and STDERR).
To set them, use std_set_send_func and std_set_get_func. Generally this happens
in the "platform_init" function (see "platform interface.txt") to set the
initial "console" device:

```
(BEGIN src/platform/at91sam7x/platform.c)
int platform_init()
{
.........................
  // Set the send/recv functions
  std_set_send_func( uart_send );
  std_set_get_func( uart_recv );
.........................
}
(END src/platform/at91sam7x/platform.c)
```

The code above makes it possible for you to use the UART as the Lua console,
thus being able to use the standard Lua interpreter (for example) via your
serial connection.
If you need another console device, just call

```
std_set_send_func/std_set_get_func
with the appropriate function pointer.
```

eLua platform interface

==

This document describes the "platform" interface in eLua. Its purpose is to ease
the task of porting eLua to a new platform, as well as having a uniform layer
for accesing peripherals (such as PIO, UART, SPI ...) on all platforms.

The definitions of the functions shown here are in the "inc/platform.h" file.
Some of the functions are required, others are optional; see also the "adding
a new platform.txt" and "platform modules.txt" files for additional information.
Also, for each function or function group, the name of the module(s) that use
it (if any) is specified. If other part of the code uses the module, a
"ALSO USED BY" line will be present in the module description.

===

```
int platform_init();
```

TYPE: REQUIRED
USED BY MODULE: N/A

PURPOSE: platform-specific initialization (this is a good place to initialize
the platform CPU, as well as the CPU peripherals, like the UART).
RETURNS: PLATFORM_OK or PLATFORM_ERR (if PLATFORM_ERR is returned the program
blocks in an infinite loop).

===

```
void* platform_get_first_free_ram( unsigned id );
void* platform_get_last_free_ram( unsigned id );
```

TYPE: REQUIRED
USED BY MODULE: N/A

PURPOSE: returns the first and the last free RAM address; the space between them
will be used for the system heap. 'id' is a memory space identifier.

This can be
used if there is more than one RAM memory available in the system, and their
address ranges do not overlap. For example, one can have a CPU with internal RAM
(a very common case) but also an external RAM chip. In this case there are two
memory spaces, the first one being the internal RAM and the other one the
external RAM. While each of them in part is contiguous, they are generally not
contigous to each other in the system address space, so they must be treated as
two separate address spaces. If the multiple allocator (see building.txt) is
used you can define as many memory spaces as you wish in your system, the
allocator will make sure to use all of them.
If the system RAM exists in a single memory space (for example the internal RAM
on the MCU) the CPU's stack pointer should be set at the end of the RAM at
startup. Thus, the first free ram will start right after the data/bss sections,
and the last free RAM is the last physical address of RAM minus the size of the
stack. The heap and the stack will grow on opposite directions (upward/downward)
and the heap will stop if asked to grow "over" the stack.
If the MCU has both internal RAM and external RAM, a simple arrangement is to
place the CPU stack at the end of the internal RAM and the heap in the external
memory (which is generally much larger than the MCU's internal memory). Another
arrangement is to use the multiple allocator and the memory space id as
described above.

===

```
int platform_pio_has_port( unsigned port );
const char* platform_pio_get_prefix( unsigned port );
int platform_pio_has_pin( unsigned port, unsigned pin );
pio_type platform_pio_op( unsigned port, pio_type pinmask, int op );
```

TYPE: OPTIONAL
USED BY MODULE: pio

PURPOSE: PIO operations. eLua defines a number of "virtual ports", each one 32
bits in size, as shows in "inc/platform.h". But since it somehow needs to map
these virtual ports to physical ports, it will ask the platform if a port is
physically present (via platform_pio_has_port) and also if a bit (a "pin") in
the port is physically present (via platform_pio_has_pin).
platform_pio_get_prefix gets a port number and return the "port name" as defined
in the device datasheet. Some devices use PA0, PA1, others simply P0, P1. This
is what this function is expected to return.
The platform_pio_op function is the one that does the actual work with the PIO
subsystem. It receives an operation id ("op") as well as a mask ("pinmask") to
which the operation applies. The possible operations are shown in the 'enum'
below (taken from "inc/platform.h"):

```
(BEGIN inc/platform.h)
enum
{
  // Pin operations
  PLATFORM_IO_PIN_SET,                          // Set pin(s) to 1
  PLATFORM_IO_PIN_CLEAR,                        // Set pin(s) to 0
  PLATFORM_IO_PIN_GET,                          // Get value of pin
  PLATFORM_IO_PIN_DIR_INPUT,                    // Configure pin(s) as
input
  PLATFORM_IO_PIN_DIR_OUTPUT,                   // Configure pin(s) as
output
  PLATFORM_IO_PIN_PULLUP,                       // Enable pullups on the
pin(s)
  PLATFORM_IO_PIN_PULLDOWN,                     // Enable pulldowns on
the pin(s)
  PLATFORM_IO_PIN_NOPULL,                       // Disable all pulls on
the pin(s)
  // Port operations
  PLATFORM_IO_PORT_SET_VALUE,                   // Set port value
  PLATFORM_IO_PORT_GET_VALUE,                   // Get port value
  PLATFORM_IO_PORT_DIR_INPUT,                   // Configure port as
input
  PLATFORM_IO_PORT_DIR_OUTPUT                   // Configure port as
output
};
(END inc/platform.h)
```

```
==================================================================================
int platform_spi_exists( unsigned id );
u32 platform_spi_setup( unsigned id, int mode, u32 clock, unsigned
cpol,
                        unsigned cpha, unsigned databits );
spi_data_type platform_spi_send_recv( unsigned id, spi_data_type data
);
void platform_spi_select( unsigned id, int is_select );

TYPE: OPTIONAL
USED BY MODULE: spi

PURPOSE: SPI operations. eLua defines 4 "virtual" SPI interfaces. The
function
platform_spi_exists() gets an identifier from 0 to 3 and returns 1 if
the SPI
interface with the given identifier exists on the target machine, 0
ohterwise.
platform_spi_setup() is called to configure the SPI interface with
the given
parameters, returning the actual clock that was set for the
interface. The
actual data transfer is done by calling platform_spi_send_recv(),
which executes
a SPI "cycle" (send one byte, receive one byte). Finally,
platform_spi_select()
is used to set the state of the SPI SS (slave select) pin, if the
target's SPI
interface provides this functionality.

==================================================================================

int platform_uart_exists( unsigned id );
u32 platform_uart_setup( unsigned id, u32 baud, int databits, int
parity,
                         int stopbits );
void platform_uart_send( unsigned id, u8 data );
int platform_uart_recv( unsigned id, unsigned timer_id, int
timeout );

TYPE: OPTIONAL
USED BY MODULE: uart
ALSO USED BY: XMODEM, TERM over UART

PURPOSE: UART operations. eLua defines 4 "virtual" UART interfaces.
The function
platform_uart_exists() gets an identifier from 0 to 3 and returns 1
```

if the UART
interface with the given identifier exists on the target machine, 0
ohterwise.
platform_uart_setup() is called to configure the SPI interface with
the given
parameters, returning the actual baud that was set for the interface.
The
actual data transfer is done by calling platform_uart_send to send a
byte, and
platform_uart_recv to receive a byte. The receive function has a
timeout than
can take different values:

- timeout == 0: receive without waiting for data. If a data byte is
available
  return it, otherwise return -1.
- timeout == PLATFORM_UART_INFINITE_TIMEOUT: wait until a data byte
is available
  and then return it. This will block indefinetely if no data is
available.
- timeout > 0: if a data byte is available in the give time
(expressed in us)
  return id, otherwise return -1.

===

```
int platform_timer_exists( unsigned id );
void platform_timer_delay( unsigned id, u32 delay_us );
u32 platform_timer_op( unsigned id, int op, u32 data );
u32 platform_timer_get_diff_us( unsigned id, timer_data_type end,
                                timer_data_type start );
```

TYPE: OPTIONAL
USED BY MODULE: tmr, uart (for receive with timeout)
ALSO USED BY: XMODEM, TERM over UART

PURPOSE: timer operations. eLua defines 16 "virtual" timers. The
function
platform_timer_exists() gets an identifier from 0 to 15 and returns 1
if the
timer with the given identifier exists on the target machine, 0
otherwise.
platform_timer_delay() will block the execution for the specified
number of
microseconds, and platform_timer_get_diff_us() gets two timer values
and returns
the time difference (in microseconds) between them.
platform_timer_op() executes the specified operation on the givem

timer. The
operations are defined in an enum from inc/platform.h:

```
(BEGIN inc/platform.h)
// Timer operations
enum
{
  PLATFORM_TIMER_OP_START,              // Start the timer
  PLATFORM_TIMER_OP_READ,               // Read the value of timer
  PLATFORM_TIMER_OP_SET_CLOCK,          // Set the clock of the timer
  PLATFORM_TIMER_OP_GET_CLOCK,          // Read the clock of the
timer
  PLATFORM_TIMER_OP_GET_MAX_DELAY,      // Get the maximum achievable
delay
  PLATFORM_TIMER_OP_GET_MIN_DELAY       // Get the minimum achievable
delay
};
(END inc/platform.h)
```

===

```
int platform_pwm_exists( unsigned id );
u32 platform_pwm_setup( unsigned id, u32 frequency, unsigned duty );
u32 platform_pwm_op( unsigned id, int op, u32 data );
```

TYPE: optional
USED BY MODULE: pwm

PURPOSE: PWM operations. eLua defines 16 "virtual" PWM blocks. The
function
platform_pwm_exists() gets an identifier from 0 to 15 and returns 1
if the PWM
block with the given identifier exists on the target machinem, 0
otherwise.
platform_pwm_setup() is called to configure the SPI interface with
the given
frequency and duty cycle (the duty cycle is a number from 0 to 100
representing
the duty cycle in percents).
Finally, platform_pwm_op() implements PWM specific operations. They
are all
defined in an enum from inc/platform.h, shown below:

```
(BEGIN inc/platform.h)
// PWM operations
enum
{
  PLATFORM_PWM_OP_START,                // Start the PWM block
```

```
  PLATFORM_PWM_OP_STOP,                 // Stop the PWM block
  PLATFORM_PWM_OP_SET_CLOCK,            // Set the base clock of the
PWM block
  PLATFORM_PWM_OP_GET_CLOCK             // Get the base clock of the
PWM block
};
(END inc/platform.h)
```

==

```
void platform_cpu_enable_interrupts();
void platform_cpu_disable_interrupts();
u32 platform_cpu_get_frequency();
```

TYPE: optional
USED BY MODULE: cpu

PURPOSE: CPU interfacing. It allows the user to control some of the CPU functions
directly from Lua. platform_cpu_enable_interrupts() enables the CPU interrupts
(globally), while platform_cpu_disable_interrupts() disables them.
platform_cpu_get_frequency() returns the CPU "core" frequency in Hz.

==

```
void platform_eth_send_packet( const void* src, u32 size );
u32 platform_eth_get_packet_nb( void* buf, u32 maxlen );
void platform_eth_force_interrupt();
u32 platform_eth_get_elapsed_time();
```

TYPE: optional
USED BY MODULE: net, also used by the generic TCP/IP support

PURPOSE: network support. These functions are used by uIP (the TCP/IP stack of
eLua) to implement TCP/IP services on top of the Ethernet ones (for platforms
that have an integrated Ethernet controller, or are using an external Ethernet
controller).
platform_eth_send_packet() sends the packet pointed by "src" with size "size"
over the network.
platform_eth_get_packet_nb() reads an Ethernet packet in "buf", without
exceeding "maxlen" of data. If an Ethernet packet is not available when this

function is called, it returns 0 immediately (non-blocking receive), otherwise
it returns a negative integer if the packet size is too large or the length of
the packet if it fits in "maxlen" bytes.
platform_eth_force_interrupt() is used to force an Ethernet receive interrupt.
This is needed because uIP's processing function is called from this Ethernet
interrupt handler alone.
platform_eth_get_elapsed_time() will return the approximate time (in ms) that
passed since the last call to platform_eth_get_elapsed_time(). It is used by uIP
to process periodic events.

For an example of how these functions should be implemented, take a look at the
LM3S backend (src/platform/lm3s/platform.c)

eLua platform modules
===

(NOTE: after reading this, check also the "generic_modules.txt" file to
learn about the generic (not platform specific) modules from eLua).

In order to make eLua usable on different platform, eLua provides a number of
"platform modules" that link the language with the hardware platform. They're
mainly tied up with the platform peripherals (PIO, UART, SPI and others).
They are loaded when Lua starts (just like the "standard" modules like os, math,
string).

All the platform modules have two parts: the generic part (the one that is
exposed directly to Lua and it's supposed to be platform independent) and the
platform specific part (the one that links the module operations to actual
hardware operations). Consequently, when adding a new platform, one doesn't need
to rewrite the whole module, just the platform-dependent part. The "platform
interface.txt" file shows the conenction between platform modules and platform
interface functions. For example, the "pio" module (src/modules/pio.c) needs 3
functions for interfacing with a specific platform: platform_pio_has_port,
platform_pio_has_pin and platform_pio_op.

All the modules are located in the "src/modules" directory. Besides their
actual implementation, the "src/modules/auxmods.h" file contains the Lua
compatible description of all the modules in the system.

Sometimes it doesn't make sense to include all the modules for a particular
platform. For example, for the i386 platform it doesn't make sense to include
the "pio" module (although this is technically possible by providing "bogus"
platform interface functions, the module won't be able to do

anything on a i386
CPU, unless you want to "emulate" it via the parallel port or some other
peripheral). To accomodate this, each platform must provide a "platform_conf.h"
(src/platform/<your platform>) which (amongst other things) lists the modules
that are used for that specific platform. For example, if we want to enable
only the PIO module for the AT91SAM7X platform, the "platform_conf.h" file
would look like this:

```
(BEGIN src/platform/at91sam7x/platform_conf.h)
// Auxiliary libraries that will be compiled for this platform

#ifndef __PLATFORM_CONF_H__
#define __PLATFORM_CONF_H__

#include "auxmods.h"

...............................................................

#define LUA_PLATFORM_LIBS\
  { AUXLIB_PIO, luaopen_pio }

#endif
(END src/platform/at91sam7x/platform_conf.h)
```

On the other hand, for a platform that doesn't need to enable any modules at
all, you don't even need to define the LUA_PLATFORM_LIBS macro.
This is why some of platform functions described in "platform interface.txt" are
optional. If there are no modules that use them in one platform, then you don't
need to define them at all for that platform, not even as "bogus" functions.

===
=== The PIO module
===

The PIO module lets Lua access the programmable input/output (PIO) pins of the
microcontroller. It exposes symbolic name for ports (pio.PA, pio.PB, ... pio.PF)
and symbolic names for port pins (pio.PA_0, pio.PA_1, ...

pio.PB_30, ...). Also,
it exposes functions to access both ports and pins:

pio.setpin(value, Pin1, Pin2 ...): set the value to all the pins in the list
  to "value" (0 or 1).

pio.set(Pin1, Pin2, ...): set the value of all the pins in the list to 1.

Val1, Val2, ... = pio.get(Pin1, Pin2, ...): reads one or more pins and returns
  their values (0 or 1).

pio.clear(Pin1, Pin2, ...): set the value of all the pins in the list to 0.

pio.input(Pin1, Pin2, ...): set the specified pin(s) as input(s).

pio.output(Pin1, Pin2, ...): set the specified pin(s) as output(s).

pio.setport(value, Port1, Port2, ...): set the value of all the ports in the
  list to "value".

Val1, Val2, ... = pio.getport(Port1, Port2, ...): reads one or more ports and
  returns their values.

pio.port_input(Port1, Port2, ...): set the specified port(s) as input(s).

pio.port_output(Port1, Port2, ...): set the specified port(s) as output(s).

pio.pullup(Pin1, Pin2, ...): enable internal pullups on the specified pins.
Note that some CPUs might not provide this feature.

pio.pulldown(Pin1, Pin2, ...): enable internal pulldowns on the specified
pins. Note that some CPUs might not provide this feature.

pio.nopull(Pin1, Pin2, ...): disable the pullups/pulldowns on the specified
pins. Note that some CPUs might not provide this feature.

Port = pio.port(code): return the physical port number associated

with the
given code. For example, "pio.port(pio.P0_20)" will return 0.

Pin = pio.pin(code): return the physical port number associated with the
given code. For example, "pio.pin(pio.P0_20)" will return 20.

===
=== The SPI module
===

The SPI module lets Lua access the SPI interfaces of the target CPU. It exposes
functions for SPI setup and sending/receiving data, selecting/unselecting slave
devices, as well as different SPI specific constants.

Actual_clock = spi.setup(id, spi.MASTER | spi.SLAVE, clock, cpol, cpha,
  databits): set the SPI interface with the given parameters, returns the clock
  that was set for the interface.

spi.select(id): sets the SS line of the given interface.

spi.unselect(id): clears the SS line of the given interface.

spi.send(id, Data1, Data2, ...): sends all the data to the specified SPI
  interface.

In1, In2, ... = spi.send_recv(id, Out1, Out2, ...): sends all the "out" bytes
  to the specified SPI interface and returts the data read after each sent byte.

===
=== The UART module
===

The UART module lets Lua access the UART interfaces of the target CPU. It
exposes functions for UART setup and sending/receiving data, as well as some
UART specific constants.

Actual_baud = uart.setup(id, baud, databits,
  uart.PAR_EVEN |uart.PAR_ODD | uart.PAR_NONE,

uart.STOP_1 | uart.SSTOP_1_5 | uart.STOP_2): set the UART interface with the
  given parameters, returns the baud rate that was set for the UART.

uart.send(id, Data1, Data2, ...): send all the data to the specified UART
  interface.

Data = uart.recv(id, uart.NO_TIMEOUT | uart.INF_TIMEOUT | timeout): reads a
  byte from the specified UART interface.

uart.sendstr(id, str1, str2, ...): this is similar to "uart.send", but its
  parameters are string.

==
=== The timer module
==

It allows Lua to execute timer specific operations (delay, read timer value,
start timer, get time difference).

tmr.delay(id, delay): uses timer 'id' to wait for 'delay' us.

Data = tmr.read(id): reads the value of timer 'id'. The returned value is
  platform dependent.

Data = tmr.start(id): start the timer 'id', and also returns its value at
  the moment of start. The returned value is platform dependent.

diff = tmr.diff(id, end, start): returns the time difference (in us) between
  the timer values 'end' and 'start' (obtained from calling tmr.start or
  tmr.read). The order of end/start is irrelevant.

Data = tmr.mindelay(id): returns the minimum delay (in us) that can be
  achieved by calling the tmr.delay function. If the return value is 0, the
  platform layer is capable of executing sub-microsecond delays.

Data = tmr.maxdelay(id): returns the maximum delay (in us) that can be

```
  achieved by calling the tmr.delay function.

Data = tmr.setclock( id, clock ): sets the clock of the given timer.
Returns the
  actual clock set for the timer.

Data = tmr.getclock( id ): return the clock of the given timer.

=====================================================================
=== The platform data module
=====================================================================
It allows Lua to identify the platform on which it runs.

Platform = pd.platform(): returns the platform name (f.e. LM3S)

Cpu = pd.cpu(): returns the CPU name (f.e. LM3S8962)

Board = pd.board(): returns the CPU board (f.e. EK-LM3S8962)

=====================================================================
=== The PWM module
=====================================================================

It allows Lua to use the PWM blocks on the target CPU.

Data = pwm.setup( id, frequency, duty ): sets the PWM block 'id' to
generate the
  specified frequency with the specified duty cycle (duty is an
integer number
  from 0 to 100, specifying the duty cycle in percents). It returns
the actual
  frequency set on the PWM block.

pwm.start( id ): start the PWM block 'id'.

pwm.stop( id ): stop the PWM block 'id'.

Data = pwm.setclock( id, clock ): set the base clock of the PWM block
'id' to
  the given clock. In returns the actual clock set on the PWM block.

Data = pwm.getclock( id ): returns the base clock of the PWM block
'id'.

=====================================================================
=== The CPU module
=====================================================================
```

It brings low level CPU access to Lua (read/write memory, enable/disable interrupts).

w32(address, data) : write the 32-bit data at the specified address

w16(address, data) : write the 16-bit data at the specified address

w8(address, data) : write the 8-bit data at the specified address

Data = r32(address) : reads 32-bit data from the specified address

Data = r16(address) : reads 16-bit data from the specified address

Data = r8(address) : reads 8-bit data from the specified address

cli(): disable CPU interrupts

sei(): enable CPU interrupts

Clock = clock(): returns the CPU frequency

Also, you can expose as many CPU constants (for example memory mapped registers) as you want to this module. You might want to use this feature to access some CPU memory areas (as defined in the CPU header files from the CPU support package) directly from Lua. To do this, you'll need to define the PLATFORM_CPU_CONSTANTS macro in the platform's platform_conf.h file (src/platform/<platform name>/platform_conf.h). Include all your constants in a _C(<constant name>) definition, and then build your project.
For example, let's suppose that your CPU's interrupt controler has 3 memory mapped registers: INT_REG_ENABLE, INT_REG_DISABLE and INT_REG_MASK. If you want to access them from Lua, locate the header that defines the values of these registers (I'll assume its name is "cpu.h") and add these lines to the platform_conf.h:

```
#include "cpu.h"

#define PLATFORM_CPU_CONSTANTS\
  _C( INT_REG_ENABLE ),\
  _C( INT_REG_DISABLE ),\
```

```
  _C( INT_REG_MASK )
```

After this you'll be able to access the regs directly from Lua, like this:

```
data = cpu.r32( cpu.INT_REG_ENABLE )
cpu.w32( cpu.INT_REG_ENABLE, data )
```

For a "real-life" example, see the src/platform/lm3s/platform_conf.h file.

TCP/IP in eLua
==

eLua's TCP/IP support was designed with flexibility and ease of use in mind. It
might not provide all the functions of a "full-fledged" TCP/IP stack, but it's
still fully functional and probably easier to use than a "regular" (POSIX) TCP/IP
stack. These are the services provided by the TCP/IP stack:

- a set of functions for network access (defined in inc/elua_net.h)
- a DHCP client
- a DNS resolver
- a module ("net") which can be used from Lua to access the network functions
- a Telnet miniclient, which is used to support the eLua shell via TCP/IP instead
  of serial connections.

For more details about the networking API, consult docs/generic_modules.txt.

TCP/IP configuration
==

To configure the TCP/IP subsystem, edit src/platform/[platform]/platform_conf.h
and:

1. #define BUILD_UIP to enable TCP/IP support
2. if you'll be using the DHCP client, just #define BUILD_DHCPC to build the
   DHCP client. In any case, you must also define a static network configuration:

```
  #define ELUA_CONF_IPADDR0 ... ELUA_CONF_IPADDR3 : the IP address
  #define ELUA_CONF_NETMASK0 ... ELUA_CONF_NETMASK3 : the network mask
  #define ELUA_CONF_DEFGW0 ... ELUA_CONF_DEFGW3 : the default gateway
  #define ELUA_CONF_DNS0 ... ELUA_CONF_DNS3 : the DNS server
```

  Note that you must define both BUILD_DHCPC and the ELUA_CONF_* macros. If the
  DHCP client fails to obtain a valid IP address, the static configuration will
  be used instead. To use only the static configuration (and make the eLua image

size a bit smaller) don't define the BUILD_DHCPC client.

3. #define BUILD_DNS if you want support for the DNS server.
4. #define BUILD_CON_TCP if you want support for shell over telnet instead of
   serial. Note that you must NOT define BUILD_CON_GENERIC in this case.

TCP/IP implementation internals
===
The TCP/IP support was designed in such a way that it doesn't require a specific
TCP/IP stack implementation. To work with eLua, a TCP/IP stack must simply
implement all the functions defined in the inc/elua_net.h file. This allows for
easy integration of more than one TCP/IP stack. Currently only uIP is used in
eLua, but lwIP (and possibly others) are planned to be added at some point.
Another key point of the TCP/IP implementation (and of the whole eLua design
for that matter) is that it should be as platform independent as possible: write
everything in a platform-independent manner, except for some functions (as few as
possible and as simple as possible) that must be implemented by each platform).
To illustrate the above, a short overview of the uIP integration is given below.

uIP in eLua
===

uIP (http://www.sics.se/~adam/uip/index.php/Main_Page) is a minimalistic TCP/IP
stack designed specifically for resource constrained embedded systems. While the
design and implementation of uIP are an excellent example of what can be done
with a few kilobytes of memory, it has a number of quirks that make it hard to
integrate with eLua. First, it uses a callback approach, as opposed to the
sequential approach of "regular" TCP/IP stacks. It provides a "protosocket"
library that can be used to write uIP applications in a more "traditional" way,

but it's quite restrictive. So, to use it with eLua, a translation layer was
needed. It is implemented in src/elua_uip.c, and its sole purpose is to "adapt"
the uIP stack to the eLua model: implement the functions in inc/elua_net.h and
you're ready to use the stack. In this case the "adaption layer" is quite large
because of uIP's callback-based design.
To make the uIP implementation as platform-independent as possible, a special
networking layer is added to the platform interface (docs/platform_interface.txt
for details). There are only 4 functions that must be implemented by a backend
to use the networking layer. They might change as more TCP/IP stacks are added
to eLua, but probably the networking layer won't get much bigger than it is now.

For a more in-depth understanding of how the networking layer is implemented,
look at the LM3S implementation in src/platform/lm3s/platform.c.

Terminal support in eLua
==

NOTE: currently, this only works over serial connections (not over TCP/IP)

Besides standard stdio/stdout/stderr support (docs/console_input_output.txt),
eLua provides the "term" module to access ANSI compatible terminal emulators.
It is designed to be as flexible as possible, thus allowing a large number of
terminal emulators to be used.
To use the term module, remember to:

- build it (add BUILD_TERM in your build.h file)
- build its Lua binding (add AUXLIB_TERM in your platform_libs.h)

See docs/elua_components.h and docs/platform_modules.txt for details.

To use it, first call the "term_init" function:

```
(BEGIN inc/term.h)
...........................
// Terminal output function
typedef void ( *p_term_out )( u8 );
// Terminal input function
typedef int ( *p_term_in )( int );
// Terminal translate input function
typedef int ( *p_term_translate )( u8 );
...........................
// Terminal initialization
void term_init( unsigned lines, unsigned cols, p_term_out term_out_func,
                p_term_in term_in_func, p_term_translate term_translate_func );
(END inc/term.h)
```

The initialization function gets the physical size of the terminal emulator
window (usually 80 lines and 25 cols) and three function pointers:

- p_term_out: this function will be called to output characters to the terminal.
  It receives the character to output as its single parameter.
- p_term_in: this function will be called to read a character from the terminal.
  It receives a parameter that can be either TERM_INPUT_DONT_WAIT (in

which case
  the function returns -1 immediately if no character is available)
or
  TERM_INPUT_WAIT (in which case the function will wait for the
character).
- p_term_translate: this function translates terminal-specific codes
to "term"
  codes. The "term" codes are defined in an enum from inc/term.h:

```
(BEGIN inc/term.h)
...........................
  _D( KC_UP ),\
  _D( KC_DOWN ),\
  _D( KC_LEFT ),\
...........................
  _D( KC_ESC ),\
  _D( KC_UNKNOWN )
...........................
(END inc/term.h)
```

By using this function, it is possible to adapt a very large number
of "term
emulators" to eLua. For example, you might want to run eLua in a
"standalone
mode" that does not require a PC at all, just an external LCD display
and maybe
a keyboard for data input. Your eLua board can connect to this
standalone
terminal using its I/O pins, for example via SPI. By writing the
three functions
described above, the effort of making eLua work with this new type of
device is
minimal, as writing an "ANSI emulation" for your terminal device is
not hard.
For an example, see src/main.c, where these functions are implemented
for an
UART connection with a terminal emulator program running on PC.
To see what functions are exported to eLua by the "term" module
reffer to the
"generic_modules.txt" file from the docs/ directory.

The eLua shell
==
After you burn eLua to your board and you connect the board to your terminal
emulator running on the PC, you'll be greeted with the eLua shell prompt, which
allows you to:

- run 'lua' as you would run it from the Linux or Windows command prompt
- upload a Lua source file via XMODEM and execute in on board
- query the eLua version
- get help on shell usage

To enable the shell, define BUILD_SHELL in your build.h file, and also
BUILD_XMODEM if you want to use the "recv" command (see below). See
docs/elua_components.txt for more details about enabling the shell.

You'll need to configure your terminal emulation program to connect to your eLua
board. These are the parameters you'll need to set for your serial connection:

- speed 115200, 8N1 (8 data bits, no parity, one stop bit)
- no flow control
- newline handling (if available): CR on receive, CR+LF on send

After you setup your terminal program, press the RESET button on the bord.
When you see the "eLua# " prompt, just enter "help" to see the on-line shell
help:

```
eLua# help
Shell commands:
  help - print this help
  lua [args] - run Lua with the given arguments
  recv - receive a file (XMODEM) and execute it
  ver - print eLua version
  exit - exit from this shelll
```

More details about some of the shell commands are presented below.

The "recv" command
==
To use this, your eLua target image must be built with support for XMODEM (see

docs/elua_components.txt for details). Also, your terminal emulation program must
support sending files via the XMODEM protocol. Both XMODEM with checksum (the
original version) and XMODEM with CRC are supported, but only XMODEM with 128
byte packets is allowed (XMODEM with 1K packets won't work).
To use this feature, enter "recv" at the shell prompt. eLua will respond with
"Waiting for file ...". At this point you can send the file to the eLua board
via XMODEM. eLua will receive and execute the file. Don't worry when you see 'C'
characters suddenly appearing on your terminal after you enter this command,
this is how the XMODEM transfer is initiated.

The "lua" command

This allows you to start the Lua interpreter with command line parameters, just
as you would do from a Linux or Windows command prompt. This command has some
restrictions:

- the command line can't be longer than 50 chars
- character escaping is not implemented. For example, the next command won't work
  because of the \' escape sequences:

```
eLua# lua -e 'print(\'Hello, World!\')' -i
Press CTRL+Z to exit Lua
lua: (command line):1: unexpected symbol near '\'
```

However, if you use both '' and "" for string quoting, it will work:

```
eLua# lua -e 'print("Hello, World")' -i
Press CTRL+Z to exit Lua
Lua 5.1.4  Copyright (C) 1994-2008 Lua.org, PUC-Rio
Hello,World
>
```

The ROM file system
==

This is a small, read-only file system built inside eLua. It is integrated with Newlib, so you can use standard POSIX calls (fopen/fread/fwrite...) to access it. It is also accessible directly from Lua. The files in the file system are part of the eLua binary image, thus they can't be modified after the image is built. For the seame reason, you can't add/delete files after the image is built.
To use this file system:

- copy all the files you need to the romfs/ directory.
- Build eLua (docs/building.txt). As part of the build process, "mkfs.py" will be called, which will read the contents of the "romfs/" directory and will output a file that contains a binary description of the file system.
- burn your image to the target
- from your code, whenever you want to access a file, prefix its name with "/rom/". For example, if you want to open the "a.txt" file, you should call fopen like this:

```
  f = fopen( "/rom/a.txt", "rb" )
```

  If you want to execute one file from the ROM file system with Lua, simply do this from the shell:

```
  eLua# lua /rom/bisect.lua
```

  Or directly from Lua:

```
  > dofile "/rom/bisect.lua"
```

The maximum file name of a ROMFS file is 14 characters, including the dot between the file name and its extension. Make sure that the file names from romfs/ follow this rule.

Adding a new platform to eLua

If you want to add a new platform to eLua, the first thing you need to check is if the platform has enough resources to run Lua. Roughly speaking, 256k of Flash (or even 128k for the integer-only version) and 64k of RAM should be enough for a 32-bit platform. As usual, the more, the better (this is especially true for the RAM memory).
Next, check if a GCC/Newlib toolchain is available for the platform. To be more precise, the compiler doesn't really matter, as long as you're able to compile Newlib with it. If you don't, you won't be able to compile eLua. This limitation might be eliminated in future versions, but it's not a priority of the project, so don't count on it happening too soon.
After this, you need to make sure that you have a basic understanding of the platform, or at least of its initialization sequence. Most platforms require specific sequences for initializing the clock subsystem, or for disabling the watchdog timer, or for remapping the internal memory, and many others.
Fortunately the vast majority of chips manufacturers provide support packages for their CPUs, so once you download the support package and understand the initialization code, you should be safe. At the very least, you'll need:

- a "startup" sequence, generally written in assembler, that does very low level intialization, sets the stack pointer, zeroes the BSS section, copies ROM to RAM for the DATA section, and then jumps to main.
- a linker command file for GNU LD.
- the "high-level" initialization code (for example peripheral initialization).

Let's suppose that your new platform is called "foo". In order to compile eLua for foo, follow these steps:

1. create the src/platform/foo directory
2. modify the SConstruct file from the base directory to make it aware of your
   new CPU, platform and board(s). A "board" translates into a simple macro
   definition at compile time, and makes it easy to adapt your platform code for
   different situations. For example, you might have 2 boards with the same CPU,
   but different I/O pin assignments. By checking the value of the "ELUA_BOARD"
   macro in your code you can adapt it for each board.
3. you need at least 4 files (besides your platform specific files) in the
   src/platform/foo directory:

  - conf.py: this is read by SConstruct and describes how to build files for
    the platform, as well as the platform specific files. Start from an existent
    conf.py file and modify it to suit your needs, it's easier this way.
  - type.h: data types used by eLua, declared in a platform independent way.
    Again, start from an existent type.h file and modify it if needed.
  - platform_conf.h: see "platform modules.txt", "elua_components.txt" and
    "tcpip_in_elua.txt" for details

4. implement the platform interface functions (see "platform interface.txt"). By
   convention, they should be implemented in a file called "platform.c". Note that
   SConstruct defines 3 macros that might prove useful: ELUA_CPU, ELUA_PLATFORM
   and ELUA_BOARD.

5. That's it! Build (see "building.txt") and enjoy!

==
Building eLua
==
Before you start, you might want to check if the list of platform modules and
eLua components are set according to your needs. See docs/platform_modules.txt
and docs/elua_components.txt for details.

To build eLua you'll need:

- a GCC/Newlib toolchain for your target (see http://elua.berlios.de for
  instructions on how to build your own toolchain). Please note that even if
  you already have a compiled toolchain, the differences in the Newlib configure
  flags (mainly the --disable-newlib-supplied-syscalls flags) might prevent
  eLua for building properly on your machine.
- Linux. Compiling under windows should be possible, however this isn't tested.
  I'm using Ubuntu, so I'm also using "apt-get". If you're using a distro with a
  different package manager you'll need to translate the "apt-get" calls to your
  specific distribution.
- python. It should be already installed; if it's not:

```
  $ sudo apt-get install python
```

- scons. eLua uses scons instead of make and makefiles, because I find scons
  much more "natural" and easier to use than make. To install it:

```
  $ sudo apt-get install scons
```

- your toolchain's "bin" directory (this is generally something like
  /usr/local/cross-arm/bin, where /usr/local/cross-arm is the directory in which
  you installed your toolchain) must be in $PATH.
- if you're building for the i386 platform, you'll also need "nasm":

```
  $ sudo apt-get install nasm
```

For each platform, eLua assumes a certain name for the

compiler/linker/assembler
executable files, as shown below.

Tool Platform	Compiler	Linker	Assembler
ARM (all)	arm-elf-gcc	arm-elf-gcc	arm-elf-gcc
i386	i686-elf-gcc	i686-elf-gcc	nasm
Cortex-M3	arm-elf-gcc	arm-elf-gcc	arm-elf-gcc

If your toolchain uses different names, you have to modify the "conf.py" file
from src/platform/[your platform].

To build, go to the directory where you unpacked your eLua distribution and
invoke scons:

```
$ scons [target=lua | lualong]
  [cpu=at91sam7x256 | at91sam7x512 | i386 | str912fw44 | lm3s8962 |
        lm3s6965 | lpc2888 | str711fr2 ]
  [board=ek-lm3s8962 | ek-lm3s6965 | str9-comstick | sam7-ex256 | lpc-h2888 |
        | mod711 | pc]
  [cpumode=arm | thumb]
  [allocator = newlib | multiple]
  [prog]
```

Your build target is specified by two paramters: cpu and board. "cpu" gives the
name of your CPU, and "board" the name of the board. A board can be associated

with more than one CPU. This allows the build system to be very flexible. You
can use these two options together or separately, as shown below:

- cpu=name: build for the specified CPU. A board name will be assigned by the
  build system automatically.
- board=name: build for the specified board. The CPU name will be inferred by
  the build system automatically.
- cpu=name board=name: build for the specified board and CPU.

For board/CPU assignment look at the beginning of the SConstruct file from the
base directory, it's self-explanatory.

The other options are as follows:

- target=lua | lualong: specify if you want to build full Lua (with floating
  point support) or integer only Lua (lualong). The default is "lua".
- cpumode=arm | thumb: for ARM target (not Cortex) this specifies the
  compilation mode. Its default value is 'thumb' for AT91SAM7X targets and
  'arm' for STR9 and LPC2888 targets.
- allocator = newlib | multiple: choose between the default newlib allocator
  (newlib) and the multiple memory spaces allocator (multiple). You should
  use the 'multiple' allocator only if you need to support multiple memory
  spaces, as it's larger that the default Newlib allocator (newlib). For more
  information about this reffer to platform_interface.txt. The default value
  is 'newlib' for all CPUs except 'lpc2888', since my lpc-h2888 comes with
  external SDRAM memory and thus it's an ideal target for 'multiple'.
- prog: by default, the above 'scons' command will build only the 'elf' file.
  Specify "prog" to build also the platform-specific programming file where
  appropriate (for example, on a AT91SAM7X256 this results in a .bin file that
  can be programmed in the CPU).

The output will be a file named elua_[target]_[cpu].elf (and also another

file with the same name but ending in .bin if "prog" was specified for platforms
that need .bin files for programming).
If you want the equivalent of a "make clean", invoke "scons" as shown above,
but add a "-c" at the end of the command line. "scons -c" is also recommended
after you change the list of modules/components to build for your target (see
section "prerequisites" of this document), as scons seems to "overlook" the
changes to these files on some occasions.

A few examples:

```
$ scons cpu=at91sam7x256
```

Build eLua for the AT91SAM7X256 CPU. The board name is detected as sam7-ex256.

```
$ scons board=sam7-ex256
```

Build eLua for the SAM7-EX256 board. The CPU is detected as AT91SAM7X256.

```
$ scons board=sam7-ex256 cpu=at91sam7x512
```

Build eLua for the SAM7-EX256 board, but "overwrite" the default CPU. This is
useful when you'd like to see how the specified board would behave with a
different CPU (in the case of the SAM7-EX256 board it's possible to switch the
on-board AT91SAM7X256 CPU for an AT91SAM7X512 which has the same pinout but
comes with more Flash/RAM memory).

```
$ scons cpu=lpc2888 prog
```

Build eLua for the lpc2888 CPU. The board name is detected as LPC-H2888. Also,
the bin file required for target programming is generated.

Cross-compiling Lua programs

"Cross compilation" is the process of compiling a program on one system for a different system. For example, the process of compiling the eLua binary image on a PC is cross-compiling. Lua can be cross-compiled, too. By cross-compiling Lua to bytecode on a PC and executing the resulting bytecode directly on your eLua board you have two advantages:

- speed: the Lua interpreter on the eLua board doesn't have to compile your Lua source code, it just executes the compiled bytecode
- memory: this is more important. If you're exectuing bytecode directly, no more memory is "wasted" on the eLua board for compiling the Lua code to bytecode. Many times this could be a "life saver". If you're trying to run Lua code directly on your board and you're getting "not enough memory" errors, you might be able to overcome this by compiling the Lua program on the PC and running the bytecode instead.

But for this cross-compilation to work, the two Lua targets must be compatible (they should have the same data types, with the same size, and the same memory representation). This isn't completely true for Intel and ARM targets, as gcc for ARM uses a very specific representation for double numbers (called FPA format) by default, which makes bytecode files generated on the PC useless on ARM boards. To overcome this, a "Lua cross-compilation" patch was posted on the Lua mailing list a while ago, and it was further modified as part of the eLua project to work with ARM targets. This is how to use it (the following instructions were tested on Linux, not Windows, but they should work on Windows too with little or no tweaking):

- first, make sure that your PC has already a build system intalled (gcc, binutils, libc, headers...). You'll also need "scons". The good news is that you should have it already installed, since otherwise you won't be able to build even regular eLua.

- from the eLua base directory, issue this command:

```
$ scons -f cross-lua.py
```

  You should get a file called "luac" in the same directory after this.

- to compile your Lua code (in <source>.lua), issue this command:

```
$ ./luac -s -ccn float_arm 64 -o <source>.luac <source>.lua
```

  if you're using "regular" (floating point) Lua, or:

```
$ ./luac -s -ccn int 32 -o <source>.luac <source>.lua
```

  if you're using int-ony Lua.

- that's it! You can use the resulting file (<source>.luac) in two ways:

  - "recv" it (docs/the_elua_shell.txt)
  - copy it to the ROM file system (docs/the_rom_file_system.txt)