Easy 'C' wrapper generation for eLua using SWIG

A descriptive tutorial

| Device1  Revision History: Vx.y, 2009-## | |
| --- | --- |
| Previous Version: none | |
| Page | Subjects (major changes since last revision) |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

|  |
| --- |
| We Listen to Your Comments  Is there any information in this document that you feel is wrong, unclear or missing? Your feedback will help us to continuously improve the quality of this document. Please send your proposal (including a reference to this document) to:  <mcdocu.comments@infineon.com> |

Table of Contents

[1 Scripting languages 5](#_Toc339137771)

[2 An introduction to Lua 11](#_Toc339137772)

[3 The goal of eLua 14](#_Toc339137773)

[4 Wrapper generation with SWIG 17](#_Toc339137774)

[5 The SWIG-Lua interface and wrapper generation 19](#_Toc339137775)

[6 Lua tiny RAM (LTR) 43](#_Toc339137776)

[7 Example programs for C-Lua interface with SWIG 47](#_Toc339137777)

[8 Using SWIG with eLua for XMC4500 52](#_Toc339137778)

[9 Appendix 58](#_Toc339137779)

# Scripting languages

A scripting language, script language or extension language is a programming language that allows control of one or more applications and makes the compiler of the language part of the language runtime, and as a result, enables code to be self modified. "Scripts" are distinct from the core code of the application, as they are usually written in a different language and are often created or at least modified by the end-user. Scripts are often interpreted from source code or bytecode, whereas the application is typically first compiled to native machine code – (Wikipedia).

A few key characteristics of scripting languages can be listed as follows:

1. Development of applications with scripting is done by plugging existing components together.
2. Scripting languages generally favour high-level programming over execution speed.
3. Scripting languages are extensible and scalable. They are designed for extending the language model with new abstractions and for interoperating with components written in other languages.
4. Scripting languages are embeddable. It is possible to embed them into existing components or applications.
5. Scripting languages are in general interpreted and offer automatic memory management.
6. Scripting languages are dynamically and weakly typed and offer support for runtime introspection.

Scripting languages are usually very easy to pick up on as a result of their simplified syntax. A lot of the scripting languages have their core code written in ‘C’ which makes it very easy to extend the functionality of the language in a very modular and efficient approach. With scripts, the process of implementing an algorithm becomes very efficient. Scripts are also an interesting choice for the design and analysis of state machines.

Like system programming languages (such as C or Java), scripting languages are generally not categorized according to their syntax or their semantic domain, but according to the language constructs (or features) they offer (or do not offer). For example, depending on the provided features, a scripting language is better suited for text-processing (e.g. Perl) than for building graphical user interfaces.

There are essentially two important concepts which are of importance for scripting languages:

1. **Encapsulation and wiring:** In order to build an application as a composition of components, a scripting language must support some notion of components and connectors. More precisely, a scripting language must offer mechanisms for encapsulation and wiring.Besides the notion of components, a scripting language must offer a set of mechanisms which allow one to connect provided and required services of corresponding components. Such mechanisms can be as “low-level" as a function call or as higher-level as the pipe-operator in the Bourne Shell language. An important aspect of encapsulation and wiring is whether a language is compositionally complete (is it possible to encapsulate a composition of components as a composite component). For example, a Bourne Shell script can be used as a component of another shell script.
2. **External interoperability:** In order to use components not written in the language itself, it is necessary that a scripting language provides features to interoperate with components written in other languages.This feature is exactly denoted as external interoperability.

Popular scripting languages include Bash, Python, Perl, Ruby, Tcl and Lua.

Let’s move on to a few advanced and interesting ways of using scripting languages.

## Using scripting languages for embedded design

Scripts are a popular choice for wide variety of applications. They contribute to most of the applications running on the web (For example, Java scripts) and myriad other system programs. Scripts usually run on bigger platforms with high processing speed and advanced memory features since they better accomadate complex system programs.

Can scripts ever be used on smaller devices like micro-controllers? Are they fit enough to handle a language interpreter?

The world’s first single-chip microprocessor was the 4-bit Intel 4004 released in 1971. From there on, the design of micro-controllers is evolving by the day. For a long time, 8 bit and 16 bit micro-controllers have been extremely popular among hobbyists and professionals on account of their performance and low cost. Nevertheless, such micro-controllers have very limited resources on the chip. On an average, these devices have not more than 16K of flash memory and a kilo byte of ram (data) memory! They are also very slow in processing when compared to their 32 bit counterparts. This observation illustrates that it is not viable to run a complex system program (such as a language interpreter) on such devices. (With optimization techniques, it is theoretically possible to run such programs on smaller devices only void of any practical significance).

The time has now changed. 32 bit micro-controllers are ruling the embedded market. They are simply designed for better speed and performance. In the 32 bit domain, many micro-controllers typically have a flash memory in the range of 64 kilo bytes to 128 kilo bytes and 32 kilo bytes to 64 kilo bytes of ram (data) memory. With such advanced micro-controllers being sold at the price of their 8 bit counterparts, it seems logical to move to the 32 bit world for embedded design. Such micro-controllers can easily accomadate huge programs by crunching the numbers at faster speeds.

The world has turned around. It is certainly possible to run scripts on 32 bit micro-controllers maintaining practical significance.

## A theory of how it works

Most scripting languages are interpreted. This implies that the construct of the language includes a system program called an ‘interpreter’. The interpreter interprets the scripts or code in another format, usually called the ‘bytecode’. The target can then ultimately run the bytecode. If a smaller device such as a micro-controller can accommodate a language interpreter, it is possible to run the script directly on the target. The language interpreter can therefore be cross-compiled for the target device. This accomplished, a magical world, full of endess possibilities exist to do interactive embedded development.

**Trivia:** Interpreter firmware is readily available for some of the 8 bit microcontrollers. For example, BASIC on the early microcontrollers Intel 8052,BASIC and FORTH on the Zilog Z8 as well as some modern devices. Typically these interpreters support interactive programming.

**Note:**

BASIC and FORTH are not scripting languages.They are imperative programming languages. For information in imperative programming visit: <http://en.wikipedia.org/wiki/Imperative_programming>

## Scripting applied in practice in the embedded domain

Scripting, in general has been in existence as a versatile tool for rapid application development. It has heavily influenced computation on the internet. Myriad system programs also use scripts. A recent, yet nonchalant application of scripting has also occurred in the embedded world. A lot of fancy, practical applications have been developed so far with the application of scripting in the embedded domain.

A few companies have realized the positive aspects of scripting language in the embedded domain and have used languages like Python (it is in the maturing stage) and Lua.

Take a look at a few interesting links for information on the embedded products that have been developed with scripting.

Python (In the process of maturing):

1. Overview on embedded Python: <http://groups.google.com/group/python-on-a-chip/web/list-of-small-python-implementations?pli=1>
2. Nintendo: <http://disinterest.org/NDS/Python25.html>

**Note:**

Python cannot be used in deeply embedded system due to memory constraints.

Lua (Full fledged and supported by community):

1. ATE (Automatic test equipment):<http://www.cabletest.com/pdf/LuaInATE.pdf>
2. Wireless Com modules: <https://www.sierrawireless.com/Newsroom/newsreleases/2007/12-06-wavecom_slashes_time_to_develop_m2m_applications_by_offering_open_source_lua_plug_in_for_free.aspx>
3. Smart, programmable LCD: <http://store.earthlcd.com/ezLCD107>
4. Home automation: [http://www.netstreams.com/resproducts.php?ID=2&PId=134#](http://www.netstreams.com/resproducts.php?ID=2&PId=134)
5. Embedded web server: <http://industrial-embedded.com/real-server-pages-embedded-systems>
6. Keithley Series 2600A System SourceMeter: <http://www.metrictest.com/catalog/brands/keithley/kei_2600a.jsp>

With an increasing magnitude of applications being scripted on embedded devices, there is a very high possibility of scripting languages occupying a hot spot in the embedded market for a selective set of system applications in the near future.

The increase in hardware performance over the past few years has promoted an increase in the power and sophistication of scripting languages which, unlike conventional programming languages, can even have certain security features built-in. Nevertheless, it is important for us, as system engineers to understand the advantages and disadvantages of using scripts. Let’s add up the points.

## Pros and cons

**Advantages:**

The design of the hardware is becoming very complex these days. With increasing levels of complexity of the hardware, system programmers have moved on to more efficient ways of describing and writing program frames. The process of embedded design is greatly simplified and made efficient with the ‘C’ language. The ‘C’ language is widely used for embedded development. Gone are the days when the firmware was completely written in assembly instructions. (Combinations of ‘C’ and assembly instructions are used only for critical tasks. This is called ‘inline assembly’ code).

Sometimes, even with ‘C’ it can be quite complex to test micro-controller peripherals. For instance, the ADC module or the CAP-COM module- To make such peripherals work, one must configure the registers appropriately. Flags must be set and appropriate interrupt handling must be taken into consideration.

**Wouldn’t scriping be an elegant approach to counter this problem?**

A scripting language is a wonderful tool to perform system testing and maintainence. With scripts, it is possible to write more abstract and self-adapting, low-level drivers for hardware modules. Scripting languages are fairly high level languages. Embedded development can henceforth be done in a platform independent manner. (There can be few exceptions). It also serves as a simple tool to understand the architecture of the processor. Testing, prototyping and maintainence can become significantly simple with the use of scripts.

Reasons for why scripting can be applied for micro-controller testing:

1. Easy to learn and use.
2. Allows complex tasks to be performed in relatively few steps.
3. Allow the addition of dynamic properties.
4. Permits easy source code maintainence.
5. Better code readability.

**Customizable end user applications:**

The power of scripting languages can certainly be used in the cases where end user application customization is given a high priority. Think of practical applications such a requirement for complex motor control or a programmable robotic system. The end user must be able to drive the electronic system based on her requirements. Such configurable applications can certainly be hardcoded in ‘C’ but to configure the device, understanding ‘C’ becomes a requisite for end users. Such products will be difficult to handle for the customer.

The core drivers for such end product can certainly be written in ‘C’ (for example motor control) so as to maintain efficient firmware design but the other aspect of the product which requires user customization can be written with scripts. Scripts are easy to handle, write and maintain. The end user need not understand extensively the hardware system but can focus on the implementation of his customization requirements in an elegant, easy to tailor approach.

This too can be considered as a viable application of scripts.

**End user data management through automation:**

This would be an extension of the previous example. An application that requires constant logging of data from a system (For example, a sensor coupled with the ADC network) can be automated with the application of scripts. The end user can ultimately decide where the data should be logged (Is it maintained in the database, exported for print or constantly maintained on the external memory card supported by the board). If the application of scripts can become a bit complex for the end user, an entire graphical interface can be built aroud scripting which only requires only graphical configuration by the user. The underlying application of scripts can run behind the graphical interface. Ultimately, the application of scripts will only result in the simplification of the user interaction with the embedded system and will offer highly customizable features.

**Disadvantages:**

1. The main disadvantage of interpreting code is a much slower speed of program execution compared to direct machine code execution on the host CPU. (A technique used to improve performance is just-in-time compilation which converts frequently executed sequences of interpreted instruction to host machine code).
2. Lack of realtime behavior can be observed with scripting languages. Typical scripting languages use garbage collectors to automatically handle memory de-allocation. The activation and execution of garbage collectors is non-deterministic.

**Lack of support for critical and realtime applications:**

Scripting cannot be applied in applications which are are critical and need realtime behavior. The golden rule: “The more a programmer thinks in hexadecimal values, the more he will produce efficient code”. Scripting languages are interpreted. This offers a lot of advantages but also offer disadvantages. The code written with scripts run slower than code written in system oriented languages like ‘C’. An interpreter will interpret code a line at a time. (Runtime behavior) while compiled and linked code can directly run on the target. In a nutshell, critical fragments of code must be written in assembly or ‘C’. (For instance device drivers) and non critical applications (Which may or may not reside within the micro-controller) can use the power of scripting for efficient code production and maintainence.

As an example, a welding machine (Which needs to maintain user safety) cannot implement its safety features with interpreted languages such as scripts. They need to work realtime and perform with absolute accuracy. But a few other aspects of the device such as testing, maintainence ad user interaction can be done in a script base.

Ultimately, it is important to understand the requirement of a project to be able to choose tools more effectively. This assessment concludes the pros and cons section.

## The purpose of this manual (abstract)

This manual is inclined towards easy wrapper generation for ‘Lua’ with a wrapper generating tool called ‘SWIG’. It offers to explain the extension of ‘Lua’ as a language (as applicable to the embedding ‘Lua’ on smaller devices such as a micro-controller with ‘eLua’) with programmer defined ‘C’ modules. (For specific embedded development).

**Note:** “Embedded Lua” is referred to as “eLua”.

This manual covers extensively, the reason for selecting ‘Lua’, a powerful, fast, lightweight, embeddable scripting language and its application in the embedded world (eLua). It covers the importance of ‘wrappers’ and mechanisms for extending the functionality of ‘eLua’ with tools used to generate ‘Lua’ modules in the embedded domain. It also covers myiad examples demonstrating the same.

**Note:**

For more insight on interpreters, visit <http://en.wikipedia.org/wiki/Interpreter_%28computing%29>

## Prerequisite

This manual assumes the reader´s familiarity with the ‘C’ language and also a brief understanding of the syntax of the ‘Lua’ scripting language. A detailed knowledge of ‘Lua’ is not required.

Since most of the code examples used in the manual were tested on the Gnu/Linux environment, reader’s familiarity with the usage of the GNU tool chain, inclusive of the tools like ‘make’ is highly recommended. The manual will however keep the examples as generic as possible and is very much compatible with the Windows environment. This manual will not cover wrapper generation for C++ since the discussion is more inclined towards easy embedded firmware development and prototype testing in a script base.

# An introduction to Lua

Lua is a lightweight, multi-paradigm programming language designed as a scripting language with extensible semantics as a primary goal. It has a relatively simple C API compared to other scripting languages. Lua provides a small set of general features that can be extended to fit different problem types. It is a dynamically typed language intended for use as an extension or scripting language, and is compact enough to fit on a variety of host platforms. It supports only a small number of atomic data structures such as boolean values, numbers (double-precision floating point by default), and strings. Typical data structures such as arrays, sets, lists, and records can be represented using Lua’s single native data structure, the table, which is essentially a heterogeneous associative array. It is an interpreted language.

Lua implements a small set of advanced features such as first-class functions, garbage collection, closures, proper tail calls, coercion (automatic conversion between string and number values at run time), coroutines (cooperative multitasking) and dynamic module loading. By including only a minimum set of data types, Lua attempts to strike a balance between power and size.

For additional information about ‘Lua’, visit <http://www.lua.org/>

## Why Lua/eLua?

At this juncture, it is important to understand the choice of ‘Lua’ for cross embedded development. In the free world which gives system programmers access to some of the most advanced dynamic scripting languages like ‘Python’ and ‘Perl’, why is there even the slightest intention of using ‘Lua’ ?

There are a few simple answers.

1. ‘Lua’ is a tiny language with just about 8 data types. As a result, ‘Lua’ can be cross compiled for smaller devices.
2. ‘Lua’ is a dynamic language with an extensive garbage collector. This gives system programmers the freedom of overlooking the tedious and critical job of memory deallocation.
3. Despite its small size, ‘Lua’ offers multiple paradigms for progam design such as procedural, object oriented, imperative and the advanced concepts from the functional world.
4. The last and most important attribute of ‘Lua’/‘eLua’ is that it is powered by the permissive MIT free software license. The license permits reuse within proprietary software on the condition that the license is distributed with that software. This works as an advantage in the embedded world, where source code protection is a must.

(For an extensive overview of other free software lisences and their significance, visit <http://www.gnu.org/licenses/license-list.html>).

## Philosophy

As stated earlier, ‘Lua’ was designed to be a small yet fully functional and efficient language. ‘Lua’ can run on a wide variety of platforms. For example, it is possible to run the basic lua interpreter directly on a PC, without the intervention of an operating system by booting into the interpreter at startup. ‘Lua’ can also be ported on myriad micro-controllers. Extensive work is being done in porting ‘Lua’ on embedded platforms. This version of Lua is called ‘Embedded Lua’ or simply ‘eLua’. As a result of its size, it is possible to run the interpreter on the smallest of the processors.

Another important consequence from the design philosophy of the language is that it’s very simple to extend the construct of the language by building modules for it with absolute ease. Hence, it is possible to build libraries for various platforms and use them from the ‘Lua’ interpreter, specifically cross compiled for that platform.This makes it interesting to use ‘Lua’ for embedded design, testing and prototyping.

## Installation guide

Lua is free software distributed in source code supported by the MIT license. ‘Lua’ source can be downloaded from the official ‘Lua’ website <http://www.lua.org/download.html>. Binary version for Windows platform is also available.

**Windows:**

Windows users can download the binary version for their Windows box. The binary ‘.exe’ file can be downloaded from <http://code.google.com/p/luaforwindows/>. The project is called “Lua for Windows”. A lot of libraries are also included on the website. These libraries extend the functionality of ‘Lua’.

**Gnu/Linux:**

‘Lua’ is a standard tool available in the open source reporitory. One can use a package manager such as ‘apt’ to install ‘Lua’ on their Gnu/Linux box.

Run the following on bash:

raman@infineon:~/programs$ sudo apt-get install lua5.1

Bash will prompt for the admin password. Type it in and the installation will proceed automatically. After the installation is complete, you will be able to invoke the ‘lua’ command from the command line. To check if your installation was successful, type lua –v on your terminal. You will get the following response if your installation was successful. (Output can vary based on version).

Lua 5.1.4 Copyright © 1994-2008 Lua.org, PUC-Rio

## Extensible language vs Extension language

It’s important to understand the difference between *extension* and *extensible*. Lua is primarily an embedded language. That implies that Lua is not a stand-alone package, but a library that can be linked with other applications so as to incorporate Lua facilities into these applications. To interact with Lua directly, we need to a Lua interpreter which is an interface between the user and the code Lua library. To test the functionality of the language, debug or use an RPC mechanism to communicate with another device in Lua, we need this interactive interpreter program.

This ability to be used as a library to extend an application is what makes Lua an *extension language*. At the same time, a program that uses Lua can register new functions in the Lua environment; such functions are implemented in C (or another language) and can add facilities that cannot be written directly in Lua. This is what makes Lua an *extensible language*.

These two views of Lua (as an extension language and as an extensible language) correspond to two kinds of interaction between C and Lua. In the first kind, C has the control and Lua is the library. The C code in this kind of interaction is what we call *application code*. In the second kind, Lua has the control and C is the library. Here, the C code is called *library code*. Both application code and library code use the same API to communicate with Lua, the so called C API.

The C API is the set of functions that allow C code to interact with Lua. It comprises functions to read and write Lua global variables, to call Lua functions, to run pieces of Lua code, to register C functions so that they can later be called by Lua code, and so on. (Throughout this text, the term "function" actually means "function or macro". The API implements several facilities as macros).

## Wrappers

To understand the concept of wrappers, one needs to be familiar with the term ‘language binding’. Language binding is a provision created by system programmers to extend the functionality of language or the system as a whole by providing it with the facility to comprehend higher level syntax of other languages.

Wrappers are code sequences written to enable language binding facility in a system. These programs are generally available as shared libraries which are linked with the main program. This gives a programmer the power to write efficient and productive code in an entirely different language (Any other language the system was not written in) while maintaining compatibility with the underlying system layer.

## Advantages of language bindings and wrappers for dynamic and scripting languages

1. Supports system programming and integration.
2. Debugging code is time consuming. Consider the testing phase for micro-controllers. It’s hard to write hundreds of lines of code to perform the initial configuration of the registers and write a stub code in ‘C’. It’s easier to perform the monotonous task in a less rigid, dynamic description language.

# The goal of eLua

**eLua** stands for **Embedded Lua**. The project aims to offer the full implementation of the Lua programming language to the embedded world, extending it with specific features for efficient and portable embedded software development. eLua allows you to develop and run Lua programs on a wide variety of 32 bit microcontrollers. Support for 8 bit and 16 bit micro-controllers is not available due to the limited on chip flash and ram memory on such controllers. (It might however be possible to run eLua on a 16 bit micro-controller if a lot of the features of eLua are removed, though not practically viable).

eLua can be used to simplify the task of testing and prototyping during the phase of embedded development.

Some aspects of eLua:

1. Code can be written in a hardware-independent manner (As far as possible).
2. Source code portability can be maintained. It is possible to develop and deploy software directly on the target. In addition, programs can also be saved on sd/mmc cards.
3. Lua is a scripting language. It is much easier to maintain Lua code than ‘C’ code due to its fairly high level syntax. It also makes the code very readable.
4. Easy testing of micro-controller peripherals is possible.
5. eLua is free and open-source software. It uses the free MIT license.

Simply put, eLua is like regular ‘Lua’, only it runs on smaller devices such as a micro-controller. This build of Lua for micro-controllers (eLua) is cross compiled for a specific target with a specific toolchain.

Most of the code samples included in this section was tested for a specific target.

**Platform:** Luminary micro, LM3S

**Board:** EK-LM3S8962

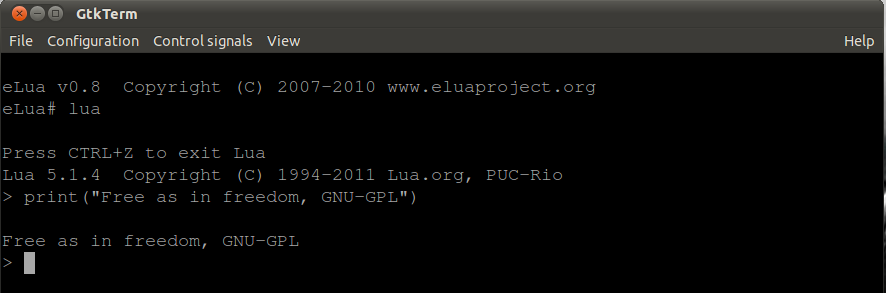
**CPU:** LM3S8962

For more information on support for other 32 bit micro-controllers, visit the eLua platform page. (<http://www.eluaproject.net/en_status.html#platforms>).

Once you have eLua running on your development board, you can write lua scripts interactively through the serial interface on the hyperterminal. If your board also supports mmc card interface, you will also be able to store lua scripts on the micro SD card and execute it directly on the target by invokving only one command!

A port of eLua for a specific cpu/board contains most of the API’s necessary to build a functional firmware. This layer provides functions which work in a target independent manner, hence making porting of an existing program to another target smooth and easy. A port of eLua also contains certain board specific functions. For example, the implementation of OLED and push buttons depend on the design of the development board. Controlling such board specific devices is managed by carefully cross-compiling eLua for that specific board. It is also not very hard to run eLua on custom boards. (A working knowledge of ‘C’ and ‘Lua’ is required).

Given below is a sample screen-shot of the eLua terminal, accessed through the serial interface. The screenshot also shows the output of the traditional “Hello world” program.



This documentation focuses on extending the basic functionality of eLua by interfacing it with wrappers that can be generated with automatic wrapper generation tools. So, thinking about it, low level system drivers for hardware can be written in ‘C’ and wrapper generation tools can be used to generate wrappers for them. These can then be called from ‘Lua’ running on the target device. This simplifies the process of development of a prototype for a certain project, extensively minimizing the time consumption.

**For standard examples and tutorials on how to get started with eLua programming, visit the link online.** <http://wiki.eluaproject.net/Projects>

**Note:**

These scripts can either be written directly on the eLua terminal or can be saved as a lua file on the mmc card. Scripts can be called from the eLua terminal as eLua# lua /mmc/your\_file.lua

## Advantages

**eLua** allows you to develop and run Lua programs on a wide variety of micro-controllers.

Some aspects of **eLua** are:

1. Code for products can be made as hardware-independent as possible. An upgrade or a complete change in hardware in the future can save time and money.
2. Source code portability is an excellent advantage while developing in eLua. With a fully functional Lua interpreter and a dedicated shell on the microcontroller, it becomes very easy to test small code snippets and port them to radically different platforms.
3. Prototyping and experimenting on a Rapid Aplication Develop model is possible with eLua. Testing of ideas and code directly on the target platforms can be done very easily and also at no cost.
4. It is highly possible to add user configuration and scripting capabilities to projects, making them adaptable to the ever changing contexts of industrial processes and customer needs.
5. **eLua** is free and open-source software and is promoted very well. The MIT licence (the same as Lua’s) allows the possibility of using **eLua** in commercial and private-code products as well.

## Disadvantages

1. Not suitable for hard real time system. (Due to the nature of support for garbage collection)
2. Slower code execution when compared to C.
3. Since Lua is weakly typed, it makes it more difficult to debug the program.

## Support

eLua is supported by dedicated free and open source community.

(<https://lists.berlios.de/mailman/listinfo/elua-dev>).

The official IRC channel for eLua (#elua) can be found on <http://freenode.net/>

Please visit the official eLua website for more information. (<http://www.eluaproject.net/>).

# Wrapper generation with SWIG

SWIG (Simplified Wrapper and Interface Generator) is a software development tool for building scripting language interfaces to C and C++ programs. Originally developed in 1995, SWIG was first used by scientists in the Theoretical Physics Division at Los Alamos National Laboratory for building user interfaces to simulation codes running on the Connection Machine 5 supercomputer. In this environment, scientists needed to work with huge amounts of simulation data, complex hardware, and a constantly changing code base. The use of a scripting language interface provided a simple yet highly flexible foundation for solving these types of problems. SWIG simplifies development by largely automating the task of scripting language integration--allowing developers and users to focus on more important problems. Although SWIG was originally developed for scientific applications, it has since evolved into a general purpose tool that is used in a wide variety of applications--in fact almost anything where C/C++ programming is involved.

SWIG is a software development tool that simplifies the task of interfacing different languages to C and C++ programs. In a nutshell, SWIG is a code generator that takes C/C++ declarations and creates the wrappers needed to access those declarations from other languages including including Perl, Python, Tcl, Ruby, Guile, and Java. SWIG normally requires no modifications to existing code and can often be used to build a usable interface in only a few minutes.

Possible applications of SWIG include:

1. Building script interfaces to existing C programs.
2. Rapid prototyping and application development.
3. Interactive debugging of a software system with stub code.
4. Reengineering or refactoring of legacy software into a scripting language components.
5. Interfacing a graphical user interface.
6. Testing of C libraries and programs (using scripts).
7. Interfacing high performance ‘C’ modules for scripting languages.

**Note:** SWIG is a command line tool.

## Installation guide

SWIG is free and open source software supported by the GNU GPL license. The original tool can be downloaded from [http://www.SWIG.org/download.html](http://www.swig.org/download.html). We have also developed LSTR (Lua SWIG tiny ram) which generates LTR compatible wrappers for smaller devices running eLua. The developmental source code can be downloaded from <https://github.com/theorist-ludwig>. A precompiled source package is distributed with this manual. You can also view the readme file for installation on Windows.

Windows:

To compile LSTR from source for windows, make sure you have maintained POSIX compliance by installing MinGW and MSYS. Copy the source a suitable directory.

1. Run the configure script with the following switch.

[raman@infineon:~/SWIG2.0.4$](mailto:raman@infineon:~/swig2.0.4$) ./configure --without-pcre

This procedure will generate the Makefiles needed to build LSTR.

2. Run make.

[raman@infineon:~/SWIG2.0.4$](mailto:raman@infineon:~/swig2.0.4$) make

This will compile the source code and generate SWIG.exe in the same working directory.

NOTE: Regular SWIG downloaded from the official site/mirror will not directly compile with the gcc toolchain for windows. A little tweaking has to be done to get it to compile without errors or warnings. The OCAML module is prone to a few issues. (Not tested). It might be fixed in the next LSTR release.

Gnu/Linux:

SWIG is a standard tool available in the open source reporitory. One can use a package manager such as ‘apt’ to install SWIG on their Gnu/Linux box.

Run the following on bash:

raman@infineon:~/programs$ sudo apt-get install SWIG

Bash will prompt for the admin password. Type it in and the installation will proceed automatically. After the installation is complete, you will be able to invoke the ‘SWIG’ command from the command line. To check if your installation was successful, type SWIG –version on your terminal. You will get the following response if your installation was successful.

SWIG Version 1.3.40

Compiled with g++ [i686-pc-linux-gnu]

Building SWIG from source:

Download the source from the link given above and follow the given instructions.

## Untar the source package.

1. ‘cd’ into the folder.
2. [raman@infineon:~/programs/SWIG-2.0.3$](mailto:raman@infineon:~/programs/swig-2.0.3$) ./configure

This program will initially configure the installation, look for dependancies and finally generate a ‘Makefile’. One can confirm this by issuing ‘ls’ on bash. (The configuration may take a while).

1. [raman@infineon:~/programs/SWIG-2.0.3$](mailto:raman@infineon:~/programs/swig-2.0.3$) make
2. [raman@infineon:~/programs/SWIG-2.0.3$](mailto:raman@infineon:~/programs/swig-2.0.3$) sudo make install
3. [raman@infineon:~/programs/SWIG-2.0.3$](mailto:raman@infineon:~/programs/swig-2.0.3$) make clean (If needed).

This completes the build process for SWIG.

You can now issue SWIG –version on the terminal to confirm successful installation.

For troubleshooting/problems/bugs, check on the SWIG user group or the official SWIG documentation.

## Is SWIG the only available tool?

SWIG certainly is not the only tool available to generate wrappers for ‘Lua’. There are myriad other applications which promise the same. One other competitive software product when it comes to ‘Lua’ and wrapper generation is ‘Tolua’. (<http://www.tecgraf.puc-rio.br/~celes/tolua/>).

Based on experience and exhaustive research, it was found that SWIG did a far superior job at generating wrappers for ‘Lua’, just given ‘C’ code than toLua. toLua is a smaller version of SWIG (SWIG supports wrapper generation for a lot of other scripting languages like Python and Tcl) which supports only ‘Lua’. Nevertheless, toLua fails to support a lot of the complex ‘C’ datatypes that SWIG supports. For instance, SWIG implements a feature called ‘Typemaps’ which lets programmers configure return values, input to a certain function in the ‘Lua’ world and to carefully control outputs of a function in ‘Lua’. Such customizable features are not supported by Tolua.

Note: The above declaration of SWIG and Tolua are to be considered valid for SWIG version: SWIG-2.0.4 and Tolua v. 2.0. Both the tools are free and open source softwares and have an active community. Bug fixes, improvements and new features may have (or may not) been introduced in the newer version of Tolua which have not been tested or monitored.

## Advantages of using SWIG for wrapper generation

1. Effective wrapper code generation.
2. Extensive support provided for traditional ‘C’ datatypes.
3. Support for advanced user defined datatypes and data structures.
4. Easy interfacing of wrapper with ‘Lua’.
5. Very wide and active free and open source community.

## Disadvantages of using SWIG

1. Huge wrapper code size.
2. Wrapper generated is not highly optimized.
3. Support for advanced features like ‘C’ function pointers are yet to be implemented.
4. Regular SWIG does not support LTR compatible wrapper generation

## Support

SWIG is free and open source software. It has a very active community. For additional information, do an extensive search on [http://www.SWIG.org/](http://www.swig.org/).

For installation problems and bugs, post requests on the mailing list. ([http://www.SWIG.org/mail.html](http://www.swig.org/mail.html)).

This section concludes the discussion of SWIG. Let us move on to concrete examples by implementing a test module through wrapper generation for ‘Lua’ using SWIG.

# The SWIG-Lua interface and wrapper generation

Now that we’ve understood the concept of wrapper generations it’s time to put all we’ve seen so far to test. (Make sure you’ve installed all the tools correctly. If not, do a quick recheck on this manual for installation procedures and get back to this section).

## ‘Hello world’ of wrapper generation

Let us write a program to print the ubiquitous “Hello world” on the screen. Here’s the twist. The code is actually written in ‘C’ but will be called from within Lua. So, a Lua script will ultimately call the ‘C’ function.

Break out your favourite text editor and try out the following procedure.

1. Write the ‘C’ program:

Firstly, the ‘C’ program has to be written in order to be made callable from ‘Lua’. Hence, write a simple function in a ‘C’ file. Let’s call it ‘hello.c’. So, make a folder in your project directory and proceed with the steps given below. **Save all the files in this directory.**

File: hello.c

#include <stdio.h>

#include "hello.h"

void print\_hello(void)

{

printf("\nHello world!\n");

}

File: hello.h

#ifndef HELLO\_H

#define HELLO\_H

void print\_hello(void);

#endif

The header file must be included in the ‘C’ file´. (As mentioned in the above example for the linker to link the functions appropriately so as to generate the final executable machine executable image)

1. Write an ‘.i’ file:

An ‘.i’ file contains information (Written in a specific format) for SWIG to understand how to generate a ‘Lua’ wrapper for your ‘C’ code. As we proceed, we will understand in detail, the functionality and syntax specifications of the ‘.i’ file.

File: hello.i

#include <stdio.h>

%module hello

%{

void print\_hello(void);

%}

extern void print\_hello(void);

The #include <stdio.h> is fairly obvious. It is a header file which supports your main project file (hello.c). It has to be initially included in the ‘.i’ description file. This exactly means that you have a few definitions in your ‘C’ program which are dependant on this header file <stdio.h>.

%module hello can be esoteric at first sight, but do not get threatened. %module is a directive. This is a syntax that SWIG understands. So think of the description as the ‘.i’ file description language. (It certainly is a language, only limited to this application). It only asks SWIG to use ‘hello’ for a module name. Hence, whatever name you choose here will be reflected in the file (wrapper file name) generated by SWIG. In this case, SWIG will generate a wrapper file called <hello\_wrap.c>.

%{ and %} are symbols which mark the content for SWIG. The block provides a location for inserting additional code, such as ‘C’ header files or additional ‘C’ declarations, into the generated ‘C’ wrapper code.

1. Invoke SWIG:

To invoke SWIG, (You must have already prepared your files in a directory) you must use proper command line arguments and supply the ‘.i’ file as an input to SWIG.

Before invoking SWIG, understand which switch suits your needs issue the following commands on your favourite terminal.

$ swig –lua –help

Lua options (available with –lua)

-elua - Generates LTR compatible wrappers for smaller devices running elua

-eluac - LTR compatible wrappers in **crass compress** mode for elua

$ swig –lua –elua case.i

After issuing this command, you will notice that a wrapper file has been generated. By default, the name of the wrapper file will be (module name)\_wrap.c. You can ask SWIG to change the default name for you.

Note: If there‘s a syntax in your .i file, SWIG will complain and the wrapper file will not be created. The user will be informed with a suitable error message.

1. Add entry in platform\_conf.h and auxmod.h files in the eLua platform source:

platform\_conf.h file is the platform configuration file for specific platforms. This file determines what libraries are going to be a part of the final eLua image

Add a \_ROM macro and include the details of your C library

A sample platform\_conf.h file (LM3S8962) is given below:

**// Add your library here:**

#define LUA\_PLATFORM\_LIBS\_ROM\

**\_ROM(“case“, luaopen\_case, case\_map )\ 🡪 case is the name of the module.**

\_ROM( AUXLIB\_PIO, luaopen\_pio, pio\_map )\

\_ROM( AUXLIB\_SPI, luaopen\_spi, spi\_map )\

\_ROM( AUXLIB\_TMR, luaopen\_tmr, tmr\_map )\

\_ROM( AUXLIB\_PD, luaopen\_pd, pd\_map )\

\_ROM( AUXLIB\_UART, luaopen\_uart, uart\_map )\

PWMLINE\

\_ROM( AUXLIB\_TERM, luaopen\_term, term\_map )\

\_ROM( AUXLIB\_PACK, luaopen\_pack, pack\_map )\

“case” for example, is the name of your library. SWIG will automatically generate the other fields luaopen\_can and can\_map. All you have to do is to add the following entry in the platform\_conf.h file.

1. Build eLua for your target:

Make sure you have the toolchain installed. Invoke scons from the command line (scons is a software building tool written in python).

$ scons platform=tricore boot=luarpc optram=1 prog

optram=1 is taken by default. Enables LTR (Lua tiny RAM)

prog generates an additional .elf file

boot option determines in what mode the eLua image will operate after the micro-controller is reset (Either normal eLua mode or RPC mode)

Add a \_ROM macro and include the details of your C library. If you‘re in the normal mode, you can see the eLua prompt.

eLua# lua -- Will take you into interactive Lua interpreter mode

print(case.print\_hello()) -- Will print “Hello world!”

## Yet another example on basic arithmetic and recursion

Let us write two additional functions in ‘C’ which can be called from the ‘eLua’ world. Let us call these functions print\_sum() and print\_fact().

The declarations are obvious. The former accepts two integer values and returns the sum, which can be printed directly in Lua. The latter accepts an integer and returns the factorial of the number through the fundamental principles of recursion.

This section is basically a modification of the “hello world” program we wrote in the previous section. So, open up your text editor and make the following modifications to the files in the ‘hello’ directory you created in the previous section.

1. Modify the source ‘C’ file.

File: hello.c

#include <stdio.h>

#include "hello.h"

void print\_hello(void)

{

printf("\nHello world!\n");

}

int print\_sum(int n1, int n2)

{

return n1 + n2;

}

int print\_fact(int num)

{

if(num == 0)

return 1;

else

return num \* print\_fact(num - 1);

}

Your hello.c file should look like something similar to the code segment given above. You have now written the ‘C’ modules. Let’s move on.

1. Update the ‘.i’ description file for SWIG.

File: hello.i

#include <stdio.h>

%module hello

%{

void print\_hello(void);

int print\_sum(int, int);

int print\_fact(int);

%}

extern void print\_hello(void);

extern int print\_sum(int, int);

extern int print\_fact(int);

Check on the updated file. This step is very important. Be sure to include all the declarations in the ‘.i’ description file. If not done correctly, you can run into ‘make’ problems very soon.

1. Invoke SWIG:

From now on, it is just a matter of testing. You already have built all programs necessary to build and test your ‘Lua’ modules. Now, simply invoke SWIG. It will update the **hello\_wrap.c** file.

[raman@infineon:~/programs/SWIG/hello$](mailto:raman@infineon:~/programs/swig-2.0.3$) swig –lua –elua hello.i

1. Add entry in platform\_conf.h and auxmod.h files in the eLua platform source:

**// Add your library here:**

#define LUA\_PLATFORM\_LIBS\_ROM\

**\_ROM(“case“, luaopen\_case, case\_map )\ 🡪 case is the name of the module.**

\_ROM( AUXLIB\_PIO, luaopen\_pio, pio\_map )\

\_ROM( AUXLIB\_SPI, luaopen\_spi, spi\_map )\

\_ROM( AUXLIB\_TMR, luaopen\_tmr, tmr\_map )\

\_ROM( AUXLIB\_PD, luaopen\_pd, pd\_map )\

\_ROM( AUXLIB\_UART, luaopen\_uart, uart\_map )\

PWMLINE\

\_ROM( AUXLIB\_TERM, luaopen\_term, term\_map )\

\_ROM( AUXLIB\_PACK, luaopen\_pack, pack\_map )\

1. Build eLua for your target:

$ scons platform=tricore boot=luarpc optram=1 prog

optram=1 is taken by default. Enables LTR (Lua tiny ram)

prog generates an additional .elf file

boot option determines in what mode the eLua image will operate after the micro-controller is reset (Either normal eLua mode or RPC mode)

Add a \_ROM macro and include the details of your C library. If you‘re in the normal mode, you can see the eLua prompt.

eLua# lua -- Will take you into interactive Lua interpreter mode

print(case.print\_hello()) -- Will print “Hello world!”

print(case.print\_fact(5)) –- Will print 120

Do you realize what you’ve done so far? You have completely simplified the process of writing modules for ‘Lua’ with the application of SWIG. Thinking about it from the embedded perspective, assuming we design an embedded application with ‘eLua’, this is how we can efficiently manage module building for ‘Lua’ without really having to know the internal workings of ‘Lua’ and specifically the ‘C-Lua’ API which can get quite annoying.

This is certainly not the end of our discussion. We have only begun. There are a lot of interesting ways of using SWIG to generate wrappers for ‘Lua’. We will go through most of them, one at a time.

## The ‘two language’ view of the world

When a scripting language is used to control a ‘C’ program, the resulting system tends to look as follows:

[Scripting language] 🡨🡪 [A set of ‘C’ functions]

In this programming model, the scripting language interpreter is used for high level control whereas the underlying functionality of the ‘C’ program is accessed through special scripting language "commands." The interpreter executes user commands and scripts. However, most of the underlying functionality is written in a low-level language like ‘C’. The two-language model of computing is extremely powerful because it exploits the strengths of each language. ‘C’ can be used for maximal performance and complicated systems programming tasks. Scripting languages can be used for rapid prototyping, interactive debugging, scripting, and access to high-level data structures such associative arrays.

## How a scripting language controls ‘C’

Scripting languages are built around a parser that knows how to execute commands and scripts. Within this parser, there is a mechanism for executing commands and accessing variables. Normally, this is used to implement the builtin features of the language. However, by extending the interpreter, it is usually possible to add new commands and variables. To do this, most languages define a special API for adding new commands. Furthermore, a special foreign function interface defines how these new commands are supposed to hook into the interpreter.

Typically, when one adds a new command to a scripting interpreter, two things must be done; first one needs to write a special "wrapper" function that serves as the glue between the interpreter and the underlying C function. Then you need to give the interpreter information about the wrapper by providing details about the name of the function, arguments, and so forth.

## Typical ‘C’ library interface

This section of the manual will cover various different use cases for wrapping different ‘C’ dataytpes, extended user defined datatypes, complex data structures and functions. It is very important to understand these techniques to be able to write appropriate ‘Lua’ programs through which we can access them.

These techniques ultimately give more structure to the scripts we write for the end application. ‘C’ programs support myriad data types like arrays, unions and pointers. Since, we are talking about ‘wrapping’ shouldn’t we also discuss how these ‘C’ data types are being accessed by ‘Lua’ through SWIG? This is exactly the crux of this section. Thinking about it, this technique of binding ‘Lua’ and ‘C’ adds additional flavor to the ‘Lua’ scripts. The programmer can be more descriptive with such an advanced setup else one might have to describe complex data structures (Such as a micro-controller’s interrupt register) in ‘Lua’ strictly using ‘Lua’ tables! (This is not only complex to maintain in ‘Lua’ but also highly inefficient).

So, in a nutshell, this extensive guide will teach you how to configure SWIG so as to wrap the ubiquitous, complex datatypes (which are a part of the ‘C’ construct) to access them from the ‘Lua’ world.

To do this, let us write a ‘C’ file (And associated header file) describing the different use cases. We will then start the discussion sequentially. Launch your favourite text editor try each instance of the following examples.

By now, you must have understood how to operate SWIG from command line. If not, please go back to the previous sections and get comfortable with the “hello world” section.

### Constants

1. File: case.h

#ifndef CASE\_H

#define CASE\_H

/\*\*

\* Dependancies.

\*/

#include <string.h>

#include <malloc.h>

#include <math.h>

#define PI 3.14

#define STRING "Free as in freedom, GNU-GPL V3"

const int num = 10;

static int count = 50;

1. File: case.c

#include "case.h"

. . .

**Note:** main() should not be included in the ‘C’ file (case.c) but it may however contain other definitions. You are free to modify and hack around.

Write an appropriate ‘.i’ description file to be able to access ‘const’ values defined in ‘C’ from the ‘Lua’ world. After SWIG generates the \_wrap file, use this file in the sample interpreter we built (Refer to section 5.1) and build the main interface file with the gcc toolchain.

Your sample ‘Lua’ program can be written as follows:

File: load.lua

print(case.num) –- Will print 10

print(case.count) -- Will print 50

case.count = 4 -- Initialize count to 4, Valid

print(case.count) -- Will print 4

print(case.const.PI) -- Will print 3.14

print(case.const.STRING) -- Will print “Free as in freedom, GNU-GPL V3”

case.num = 100 -- Will throw error. (Immutable property of ‘num’)

print(case.num) -- Control flow will not reach here

As you can observe, variables that were a part of the ‘C’ world can be accessed from ‘Lua’. There are however a few key points.

1. Notice that ‘num’ cannot be altered since it is declared as a constant in the ‘C’ world. SWIG will automatically figure this out and make sure that the value remains immutable even in the ‘Lua’ world.
2. ‘C’ macros however, behave a little differently in the ‘Lua’ world. They don’t behave like ‘const’ data types. This is quite straightforward in the ‘C’. If you define a value with a macro (For instance, #define PI 3.14, trying to change the value by redefinition is not permitted in ‘C’. This however when transported to the ‘Lua’ will behave like any normal variable. This means that PI can be changed to 100 for example wih the syntax case.PI = 100. Is this an advantage or disadvantage? That only depends on your philosophical bend. (This functionality can be altered in the source level. SWIG is free and open source).

Note: SWIG will put all the macro constants (#include’s) in a rotable called const. To access them, we need to specify: name\_of\_module.const.name\_of\_variable.

### Global variables

Global variable definitions work similar to constants. A global defined in the 'C' world can be accessed in 'Lua' just like 'const' variables. The only difference is that normal global variables (that aren’t constants) don't have the immutable value behavior. Such variables can be accessed from the 'Lua' world and can also be manipulated as per program requirements.

Include the variable declarations and definitions in the 'case.i' configuration file. (You may modify the existing '.i' file and just append data relevant to this case).

After updating the 'case.i' file with your new variable definitions, invoke SWIG from command line. This process should update your '.i' file. Run 'make' and update your binary image, so as to invoke your updated program.

Update files: case.i

extern char l = 'a';

extern float charge = (0.62 \* 10e+19);

extern double dat = 900;

enum lang {C, CPP, PYTHON, LISP, LUA, PASCAL};

int \*ptr = &count;

int (\*foobar)(int);

You must ensure that you’ve update your 'Lua' file. With the update, your 'case' table will have access to all your 'automatic storage class specifier' global variables from the 'C' world.

Check below for the sample 'Lua' script. Just copy this segment to your 'load.lua' file.

print(case.l) -- Will print 'a'

print(case.charge) -- Will print charge on a electron

print(case.dat) -- Will print 900

print(case.PASCAL) -- Will print 5, from the enumeration

case.l = ‘b’ print(case.l) –- Will print ‘b’

Note: You cannot alter the value of enums.

This way, other variables too can be modified. Notice the few additional variables (Such as the one involving 'C' pointers and function pointers). Dealing with such types of variables is a little different. This will be covered in the subsequent sections. It cannot be handled directly. You will need a function to manipulate such variables (Pointers in the ‘Lua’ world are taken as user data and do not have a fixed data type). Do not suppress your curiosity.

### Structures, bitfields and unions

Structures, bitfields and unions are special types in the ‘C’ language. They can be used to define objects that group different variables together. Wrapping ‘C’ code with SWIG certainly supports such types of data.

Consider the following declarations.

Update files: case.i

typedef struct point {

int x, y;

} point;

typedef struct intcon\_reg {

int gie:1;

int peie:1;

int tmr0ie:1;

int inte:1;

int rbie:1;

int tmr0if:1;

int intf:1;

int rbif:1;

} intcon\_reg;

typedef union data {

int x, y;

double z;

} data;

Let us try to wrap these types (struct, unions and bitfield) and access them from the ‘Lua’ world. The process is fairly straightforward. Whenever, you define a structure in your code and use SWIG to wrap it, SWIG will automatically figure out a way to transport struct definitions to the ‘Lua’ world.

In the case\_wrap.c file, SWIG adds a separate function which bears the name of the structure/union/bitfield you defined in ‘C’ (In this case, point(), data() and intcon\_reg()). These functions can be then be accessed from ‘Lua’. When this function is called in ‘Lua’, the memory for the structure variable is allocated. This object can then be modified with the simplest of the syntax. Update your load.lua file with the following script sequence.

Update file: load.lua

struct1 = case.point()-- Allocates memory for ‘point’ structure variable ‘struct1’

struct1.x = 100 -- Assigns value to the fields within the structure

struct1.y = 200

print(struct1.x, struct1.y) -- Will print ‘100 200’

union = case.data() -- Allocates memory for ‘point’ union variable ‘union’

union.x = 200 -- Assigns value to the field ‘x’

print(union.x) -- Will print 200

union.y = 300 -- Assigns value to the field ‘y’

print(union.x, union.y) -- Will print ‘300 300’

### Functions

In ‘C’ programming, all executable code resides within a function. A function is a named block of code that performs a task and then returns control to a caller. A function is often executed (called) several times, from several different places, during a single execution of the program. After finishing a subroutine, the program will return to the point after the call. Functions are a powerful programming tool. In this discussion, we also need to discuss how to wrap functions to make them accessibile from ‘Lua’. We will discuss functions that take in several kinds of parameters, introducing you to the paradigm of defining functions so as to enable SWIG wrapper generation for ‘Lua’.

#### Simple functions without pointer arguments

We’ve already seen the “hello world” example which defined a function that took no arguments and returned nothing. We also saw a slightly advanced version of functions to add integers and calculate factorial. We will look into them in greater detail.

##### Int print\_sum(int a, int b);

We have already seen ‘int print\_sum(int, int);’. This function takes two integer arguments. It returns the sum of both the integers. The wrapping for this function involves writing an appropriate ‘.i’ file and invoking SWIG. This example has already been discussed in the previous “hello world” section.

The most important point here is that how ‘Lua’ is able to understand the concept of integers. ‘Lua’ supports ‘numbers’. The concept of ‘int’ is unknown to ‘Lua’. You can observe how this is automatically taken care of by SWIG. It knows how to convert ‘int’ type that belongs in the ‘C’ world to ‘number’ type which is valid in the ‘Lua’ world. Hence, to define functions that accept and return integers, the programmer is quite safe while using SWIG. It is a very smooth transition.

##### int fact(int num);

This recursive based factorial too has been discussed in the “hello world” section. The same concept applied in the previous int print\_sum(int, int) example applies here. It is exactly the same, just that in this case, fact() is a recursive function. This too can be called in ‘Lua’ without any issue. So, a simple script in ‘Lua’ such as print(case.fact(5)) will print 120.

**Note:**

As already explained, SWIG will automatically convert the ‘int’ type to a ‘number’ type on ‘Lua’. Hence, with the wrapper code, ‘Lua’ will be able to understand the integer conventions and treat it appropriately in ‘Lua’. This requires no special treatment. It is silently handled by SWIG. This process is done automatically for other types too such as float and double. The programmer need not think about ‘C’ data type representation in ‘Lua’.

This concludes the SWIG wrapping for functions that don’t accept complex arguments such as user defined variables or pointers.

#### Functions with pointer type arguments

SWIG's treatment of basic datatypes was described. In particular, primitive types such as int and double are mapped to corresponding types in the target language. For everything else, pointers are used to refer to structures, arrays, and other user-defined datatypes. However, in certain applications it is desirable to change SWIG's handling of a specific datatype. For example, you might want to return multiple values through the arguments of a function. This section describes some of the techniques for doing this.

One very important procedure to work with functions involving pointers: Include the cpointer.i directive library. With this library, SWIG will be able to recognize and treat ‘C’ pointers and manipulate them from ‘Lua’. This must be included in the main ‘.i’ description file (case.i) that belongs to the project. Just include the following declaration to enable usage of pointers while wrapper generation.

%include "cpointer.i"

Your case.i file should look similar to this:

%module case

%include "cpointer.i"

%{

#include <string.h>

#include <malloc.h>

#include <math.h>

#include "case.c"

%}

With cpointer.i set in your case.i file, pointer handling will be a very smooth process. You will be able to wrap pointers without any issue using SWIG. This set, proceed with bulding with functions in ‘C’.

##### void add(double \*a, double \*b, double \*result);

void add(double \*a, double \*b, double \*result)

{

\*result = \*a + \*b;

}

From reading the source code, it is clear that the function is storing a value in the double \*result parameter. However, since SWIG does not examine function bodies, it has no way to know that this is the underlying behavior. One way to deal with this is to use the typemaps.i library file and write interface code like this:

// Simple example using typemaps (‘.i’ file)

%module example

%include "cpointer.i"

%include "typemaps.i"

%inline

%{

extern void add(double \*a, double \*b, double \*result);

%}

Just add the following statement in the ‘.i’ file for the above function definition ‘add()’. Also declare the same within the curly braces that includes the header files.

extern void add(double \*INPUT, double \*INPUT, double \*OUTPUT);

This implies that despite the fact the actual function definition in ‘C’ doesn not return a value (void), the function when called from the ‘Lua’ world can actually return a value by using this \*OUTPUT directive. This is how a pointer argument can be converted to a return value in a target language, in this case, ‘Lua’.

Note: There is however another way to specify the INPUT/OUTPUT config for the parameters of a function. Read the %apply directive in the official SWIG documentation for more information about the same.

This means that ‘add()’ as a function takes in 2 parameters, both of which are inputs in the ‘Lua’ world and the 3’rd argument (declared as \*OUTPUT) is an output in the ‘Lua’ world despite the fact that the original function definition in ‘C’ does not return any value. This way it is very possible to route data appropriately in the ‘Lua’ world by using this directive. Once this is described in the ‘.i’ file, SWIG will generated the updated wrapper ‘C’ file which can be compiled in with the main ‘Lua’ interpreter interface program.

A sample code script written in ‘Lua’ print(case.add(2, 3)) will print ‘5’. Notice that only two arguments can be supplied. If you supply 3 arguments in ‘Lua’, it will complain.

This way, most of the other practical ‘C’ function (which involve pointers) can be used within ‘Lua’ with the proper usage of ‘typemaps.i’library.

##### void add\_square(double \*a, double \*b, double \*res, double \*res\_square);

void add\_square(double \*a, double \*b, double \*res, double \*res\_square)

{

\*res = \*a + \*b;

\*res\_square = (\*a + \*b) \* (\*a + \*b);

}

add\_square() function adds two ‘C’ variables of type 'double' and returns two outputs in the ‘Lua’ world- the sum of the two numbers and the square of the sum.

In the previous example, we used the OUTPUT directive once. In this example, let us try to use the OUTPUT directive in the function 2 times. This should technically return two values in the ‘Lua’ world. (Notice that the original ‘C’ function definition for add\_square() returns no value).

Update your ‘.i’ description file (case.i) by adding the following description.

extern void add\_square(double \*INPUT, double \*INPUT, double \*OUTPUT, double \*OUTPUT);

This means that add\_square() is a function accepting 4 arguments, two of which are input parameters in the ‘Lua’ world and the last two parameters behave like output from the ‘Lua’ world.

Invoke SWIG to update the case\_wrap.c file. Modify your load.lua file and include the following ‘Lua’ script.

print(case.add\_square(2, 3)) – Will print 5 25

This concludes that a function call in the ‘Lua’ world can have multiple returns. All of this is specified in the ‘case.i’ file. Given proper descriptions, SWIG will take care of all the data handling required to represent pointers and convert them appropriately in the ‘Lua’ world.

##### int new\_add(int \*x, int \*y, int \*res);

int new\_add(int \*x, int \*y, int \*res)

{

\*res = \*x + \*y;

return \*res;

}

new\_add() function adds two ‘C’ variables of type 'int' and returns an output in the ‘Lua’ world. It also executes the possibility of an actual function return in ‘C’ coupled with an ‘OUTPUT’ in ‘Lua’. The output will contain two copies of the same result. This is just an extension of the previous example, trying the possibility of value returns from functions and also OUTPUT typemaps directive at the same time.

Update your ‘.i’ description file (case.i) by adding the following description.

extern int new\_add(int \*INPUT, int \*INPUT, int \*OUTPUT);

This means that new\_add() is a function accepting 3 arguments and returning an ‘int’ parameters in the ‘C’ world. The first two arguments are input parameters in the ‘Lua’ world and the last parameter behaves like output from the ‘Lua’ world.

Invoke SWIG to update the case\_wrap.c file. Modify your load.lua file and include the following ‘Lua’ script.

print(case.new\_add (2, 3)) – Will print 5 5

This concludes that multiple returns in the ‘Lua’ world can either be because of ‘typemaps.i’ configurations and/or an actual ‘C’ function return which is just mapped appropriately in ‘Lua’.

##### char \*my\_strcat(char \*source, char \*appstr);

char \*my\_strcat(char \*source, char \*appstr)

{

int i, j;

int len = strlen(source) + strlen(appstr);

char \*final = (char \*) malloc(len + 1);

\*(final + len) = '\0';

for(i = 0; \*(source + i); i++)

\*(final + i) = \*(source + i);

for(i = 0, j = strlen(source); \*(appstr + i); i++, j++)

\*(final + j) = \*(appstr + i);

return final;

}

Consider the simple string concatenation function, written in ‘C’. This function deals with data types that we haven’t seen so far. This function concatenates two strings and returns a pointer to the character in the ‘C’ world. (Basically a string). Let us now try to wrap this function in SWIG and access it from the ‘Lua’ world.

Both the parameters supplied to the function my\_strcat() are strictly input parameters. These input values are concatenated and returned as results. So, in this case, we must apply the INPUT typemap directive on both the parameters to the function.

Update your ‘.i’ description file (case.i) by adding the following description.

extern char \*my\_strcat(char \*INPUT, char \*INPUT);

Invoke SWIG to update the case\_wrap.c file. Modify your load.lua file and include the following ‘Lua’ script.

print(case.my\_strcat(“hello”, “world!”)) –- Will print “helloworld!”

Note that SWIG automatically figures out a way to represent a string in ‘C’ (char \*) in ‘Lua’ as ‘string’.

This feature triggers a lot of fancy ideas for designing parsers in ‘C’ and calling them in ‘Lua’. This is exactly how the ‘Lua’ construct can be expanded in general by writing modules for it in ‘C’.

**Note:**

For more information on parsing, visit: <http://en.wikipedia.org/wiki/Parsing>

##### void negate(int \*num);

negate() function negates a number. For example, the negation (indicated by ‘~’ in ‘C’) of 0xFF is 0. So, basically this function exactly does bitwise inversion (Converting all 1’s to 0 and all 0’s to 1).

Not that this function only accepts one argument which is the input for the function itself. How is it returning the ouput of the computation?

This function determines the possibility of using the same function argument for both an input and output in the ‘Lua’ world. In this case, ‘num’.

Take a look at the sample ‘C’ definition.

void negate(int \*num)

{

\*num = ~(\*num);

}

The function returns void and accepts only one parameter. Here’s the trick to configure ‘num’ as both input and output.

Update your ‘.i’ file with the declaration given below.

extern void negate(int \*INOUT);

Note that ‘INOUT’ is a typemaps directive and can be used in these cases. It instantly specifies that the parameter ‘num’ will carry the input and finally contain the output when the ‘Lua’ call returns. Invoke SWIG to update case\_wrap.c wrapper file.

Type out a simple ‘Lua’ script to perform negation and execute the main executable code of the interpreter interface.

print(case.negate(1)) – Will print ‘-2’

The output will directly be printed on the console with ‘num’ treated as an output in this case.

##### void distance(point \*p1, point \*p2, int \*res);

This example is a slightly advanced example. It involves representation of user defined data types in ‘C’ (such as structures) in ‘Lua’.

**Golden rule:** **Any data type that SWIG doesn’t understand (structure variables, unions for example) will be treated as pointers by SWIG.** So, ultimately, application with such datatypes shouldn’t be any different from what we’ve seen so far with previous examples involving pointers. A datatype such as ‘point’ (which is user defined) is represented as ‘userdata’ in the ‘Lua’ world. (Read the official ‘Lua’ document to understand the basic ‘Lua’ datatypes. <http://lua-users.org/wiki/LuaTypesTutorial>).

void distance(point \*p1, point \*p2, int \*res)

{

\*res = sqrt(pow((p2->x - p1->x), 2) + pow((p2->y - p1->y), 2));

}

**Note:** **typedef struct point point;**

This function calculates the geometrical distance between two points represented in 2 dimensions. Each point has its corresponding ‘x’ and ‘y’ coordinate. The function uses the standard ‘C’ math library for functions such as sqrt() and pow() to compute the distance with the “Distance Formula” is a variant of the “Pythagorean Theorem”.

To test this function out in ‘Lua’ first update the ‘case.i’ SWIG description file by adding the declaration given below.

extern void distance(point \*INPUT, point \*INPUT, int \*OUTPUT);

This implies that the first 2 arguments are the input points, (Each containing their ‘x’ coordinate and ‘y’ descriptions) and the output (an integer) in the ‘Lua’ world.

Invoke SWIG and update the file ‘case\_wrap.c’ after including the function definition for distance() in the case.c file.

Write the following script code in ‘load.lua’ to test the function from ‘Lua’.

struct\_1 = case.point() -- Create a ‘point’ variable ‘struct\_1’

struct\_2 = case.point() -- Create a ‘point’ variable ‘struct\_2’

struct\_1.x = 1 -- Initialize the ‘point’ variable.

struct\_1.y = 2

struct\_2.x = 7

struct\_2.y = 10

print(case.distance(struct\_1, struct\_2)) – Will print ‘10’.

Notice how ‘struct point’ is being converted to a convenient representation in ‘Lua’? There is no need for any special declaration or activation in SWIG. When you declare a structure, SWIG automatically creates wrappers (accessible through ‘case.point()’) to wrap struct point in ‘Lua’. This is a very efficient way of writing scripts in ‘Lua’. It almost seems like case.point() is put in a separate namespace called ‘case’, which is a very elegant situation for script maintainence and management.

##### point inc\_one(point \*p);

inc\_one() function increments the x and y coordinate of a point and returns the new value as a structure. The

function takes in a reference as a formal argument.

This situation is a slightly modified version of the pevious structure example since the function returns a ‘struct‘ variable in ‘Lua’ (Basically represented as userdata in ‘Lua’). It is very much possible to handle the ‘point’ user data.

Take a look at the function definition in ‘C’.

point inc\_one(point \*p)

{

point temp;

temp.x = p->x + 1;

temp.y = p->y + 1;

return temp;

}

Firstly, update the ‘.i’ description file ‘case.i’ by adding to it the following declaration.

extern point inc\_one(point \*INPUT);

This implies that the function inc\_one() accepts a ‘point’ argument as an input in the ‘Lua’ world and returns a point variable. How do we go about writing a script to access the function from ‘Lua’?

struct\_1 = case.point() -- Create a ‘point’ variable ‘struct\_1’

struct\_2 = case.point() -- Create a ‘point’ variable ‘struct\_2’

struct\_1.x = 1 -- Initialize the ‘point’ variable.

struct\_1.y = 2

struct\_2 = case.inc\_one(struct\_1) –- Increment x, y by one.

print(struct\_2.x, struct\_2.y) –- Will print ‘2 3’

The crux is very simple. Since the function returns a userdata, we only have to think of a way to accept the return variable with another instance of a similar userdata. ‘struct\_1’ occupies the same size as that occupied by the value the function call returns.

Observe the ease with which such scripts can be wriiten in ‘Lua’. It is very easy to manipulate userdata, as if it were a part of the original ‘Lua’ construct.

This section concludes the discussion of wrapping of functions which are directly supported by SWIG. The next section will discuss a few situations where SWIG doesn’t explicitly support a few other datatypes and interaction with such datatypes through SWIG needs additional description in the ‘.i’ files and a whole lot of tweaking. Make sure you’ve understood everything until this point; the subsequent sections can be slightly daunting otherwise.

#### void print\_into\_array(int arr[], int des[], int len);

print\_into\_array() function reads an input array ‘arr’ and makes a copy of it in the ‘des’ array (See function declaration). 'len' is taken as a control variable.

void print\_into\_array(int arr[], int des[], int len)

{

int i;

for(i = 1; i <= len; i++)

\*(des + i) = \*(arr + i);

}

Take a look at the function definition. We will access this function from the ‘Lua’ world.

Update your case.i file with the entry given below.

extern void print\_into\_array(int arr[], int des[], int INPUT);

Write down a sample script in ‘Lua’.

-- Declare ‘arr’ and ‘brr’ as integer arrays with size ‘5’ each

arr = case.new\_int\_array(5)

brr = case.new\_int\_array(5)

-- Call the function ‘print\_into\_array()’

case.print\_into\_array(arr, brr, 5)

-- Print the array elements (Written in ‘C’)

case.print\_array(brr, 5)

Output: 2 4 6 8 10

Evtensive support for handling ‘C’ arrays is given by SWIG. Through these functions generated by SWIG, it is possible to manipulate arrays of standard ‘C’ data types.

### Funtions not directly supported by SWIG

Wrapping for a few peculiar functions are not directly supported by SWIG. They need a few additional descriptions and directive libraries. So far, we haven’t really discussed the possibility of wrapping ‘C’ arrays with SWIG. This is not directly supported out of the box. Assuming it is anyway supported, it still cannot be directly manipulated by the usage of ‘[]’ operators in ‘Lua’, like in the ‘C’ language. The ‘[]’ operators in ‘Lua’ are associated with ‘tables’, the ubiquitous approach towards declaring and defining data structures in ‘Lua’. This feature of thinking of a ‘C’ array as a ‘table’ in ‘Lua’ is not exactly supported by SWIG wrapping. This section will guide you through wrapping arrays implicitly and use them in practical functions. It also will talk about other issues while dealing with returns of pointers to structure variables and the like.

#### void print\_array(int arr[], int len);

print\_array() function prints out an integer array on the console. It accepts the reference to the array and the length of the array. It iterates over the length and prints the array for the given length.

As mentioned earlier, it is not possible to directly manipulate arrays like ‘tables’ in ‘Lua’. We need a special directive library called carrays.i (Just like cpointer.i). When this is included, SWIG will start supporting inclusion wrapping of arrays (implicitly) by generating a few special functions (defined in your final wrapper file) to access and manipulate the array. For this, one must specify how these special functions for handling arrays must be generated by SWIG and what type of array is required to be accesses and manipulated from the ‘Lua’ world.

This can be done by including an extra line of description in the case.i file. Add this to your description file in the project directory.

%array\_functions(int, int\_array);

This means that we are requesting SWIG to create special functions (within case\_wrap.c, the final wrapper ‘C’ file) to handle an array which contains ‘int’ values (an integer array) with the name you want to associate with it (in this case, the name of the array will be ‘int\_array’).

The following is the definition for the function print\_array().

void print\_array(int arr[], int lim)

{

int i;

for(i = 1; i <= lim; i++)

printf("%d ", \*(arr + i));

}

Finally, add this declaration to your case.i description file.

extern void print\_array(int arr[], int INPUT);

This implies that print\_array() is a function that accepts an array (Note that one cannot associate an array with ‘INPUT’, ‘OUTPUT’ or ‘INOUT’ directives since it is not supported) and the length of the array which is taken as an input.

When your invoke SWIG, the case\_wrap.c file will contain functions for manipulating integer arrays, accessed through the name ‘int\_array’.

The functions used to create the array, access it and manipulate it can be accessed from the ‘Lua’ world.

Type the following on your favourite text editor and save it into your ‘load.lua’ file.

–- Initialize the array with 5 elements

arr = case.new\_int\_array(5)

–- Set the array elements

for i = 1, 5 do case.int\_array\_setitem(arr, i, i \* 2) end

-- Print the array elements (Written in ‘C’)

case.print\_array(arr, 5)

The output on the console: 2 4 6 8 10

Check on the new functions ‘new\_int\_array()’ and ‘int\_array\_setitem()’. These are functions generated automatically by SWIG to interact with ‘C’ arrays. This was because of the description we wrote in the case.i file. This is the ultimate significance of the ‘carrays.i’ directive library.

So basically, any function resembling the form name\_array\_\*() is a SWIG generated function as a result of the declaration of ‘carrays.i’ directive and the definition of the array needs, callable from ‘Lua’.

#### Int \*alloc\_int\_array(int \*size);

alloc\_int\_array() function allocates an integer array in memory. It accepts the size, allocates array and returns a reference to the allocated array in ‘C’. This function is a slightly complicated version of the ‘C’ pointer examples we have seen so far. This function when wrapped for ‘Lua’, expects to handle a reference value an treat it as a array (return value is a reference). Let us proceed and write down the code and check on its application in ‘Lua’.

int \*alloc\_int\_array(int \*size)

{

int \*arr;

arr = (int \*) malloc(sizeof(int) \* (\*size));

return arr;

}

Include the following declaration in your description file case.i.

extern int \*alloc\_int\_array(int \*INPUT);

This implies that alloc\_int\_array() is a function that returns a reference to an ‘int’ type while accepting the size as an input.

Let us also write a sample ‘Lua’ script to test this function from ‘Lua’.

-- ‘pointer’ will hold the reference to the allocated array.

pointer = case.alloc\_int\_array(5)

-- Iterare through the array and set the items using the same function used

--previously.

for i = 1, 5 do case.int\_array\_setitem(pointer, i, i \* 2) end

-- Print the array with the function already created in the previous section.

case.print\_array(pointer, 5)

Output on the console:

2 4 6 8 10

Check on how we exected a function which manipulated a reference to the newly allocated integer array. We used previously defined ‘int\_array’ functions to achieve it.

Here’s an easy trick for programmers:

1. Understand the function you have written in ‘C’. Check on what kind of parameter it returns in ‘C’.
2. Next, understand how ‘Lua’ will treat such returns. For basic datatypes, SWIG will automatically handle the conversion from ‘C’ types to appropriate ‘Lua’ types. For returns involving complex ‘C’ variables (such as references), figure out a way (in Lua) to represent such returns. In this case, the function returned a pointer to an integer (basically, an integer array). The only way to manipulate this value in ‘Lua’ is to define a type in ‘Lua’ similar to the one the function returns. So, logically, we may use a combination of functions generated by using carrays.i and a few function like print\_array() which are already wrapped. This makes it very easy to handle wrapper generation using SWIG.

**Note on memory deallocation:**

It is important to understand how to deallocate memory that was allocated. The garbage collector for Lua acts on all sorts of values that consume memory and collects them when they run out of scope. Automatic garbage collection is handled by SWIG for known types. For example, structures and other user defined datatypes registered with SWIG are handled automatically by the generation of the \_\_gc() metamethod. For types that are not known to SWIG, (For example the memory allocated in the above function) the programmers have to manually write functions to deallocate memory. These functions should be wrapped along with the allocate C functions and called from within Lua for deallocation.

#### point \*pdistance(point \*p1, point \*p2, int \*res);

point \*pdistance(point \*p1, point \*p2, int \*res)

{

\*res = sqrt(pow((p2->x - p1->x), 2) + pow((p2->y - p1->y), 2));

return p1;

}

Consider the above function definition. pdistance() function does the same job as the function 'distance()' defined in previous examples. In addition to returning the geometrical distance, it also returns a reference of the point p1. Hence, while the function is called in ‘Lua’, two values are supposed to be returned.

Calling this function in ‘Lua’ teaches us how to handle pointers to user defined data types (such as \*point) in ‘Lua’ with the appropriate handling and interaction with SWIG.

To use this function in ‘Lua’ you need to update your case.i description file. Add the following declaration.

extern point \*pdistance(point \*INPUT, point \*INPUT, int \*OUTPUT);

This implies that the first two arguments are point reference inputs to the function in ‘Lua’ and the 3’rd argument is a reference to the integer, behaving as an output.

This function can be made to work in ‘Lua’ with no modification to the existing case.i declarations. It will just work out of the box.

Append this script sequence to your ‘load.lua’ file.

struct\_1 = case.point() -- Create a ‘point’ variable ‘struct\_1’

struct\_2 = case.point() -- Create a ‘point’ variable ‘struct\_2’

struct\_1.x = 1 -- Initialize the ‘point’ variable.

struct\_1.y = 2

struct\_2.x = 7 -- Initialize the ‘point’ variable.

struct\_2.y = 10

--Create temp structure to receive the reference returned by ‘pdistance()’

temp\_struct = case.point()

temp\_struct, a = case.pdistance(struct\_1, struct\_2)

print(temp\_struct.x, temp\_struct.y)

print(a)

Output: 1 2

10

You may wonder how we can initialize ‘temp\_struct’ (a regular structure variable) to a reference value returned by pdistance() function. The trick lies here. When, case.point() is called, this infact returns creates an instance of the original ‘point’ structure. Hence, initializing it to another function that returns a reference to the same structure ‘point’ doesn’t make it illegal. This is how we can exploit this basic fact of pointer return from ‘case.point()’ and use it to elegantly.

The output shows that the function call in ‘Lua’ returns two values, one value is a reference to the ‘point’ structure and the other one is the actual computed geometrical distance between the two points.

Both these values are verified on the output.

#### return\_pointer\_distance(point \*p1, point \*p2, point \*res);

int return\_pointer\_distance(point \*p1, point \*p2, point \*res)

{

int cal;

cal = sqrt(pow((p2->x - p1->x), 2) + pow((p2->y - p1->y), 2));

res->x = p1->x;

res->y = p1->y;

return cal;

}

This function is very similar in action to the previous example, but only presented in a different way. The previous example returned a reference to a structure variable and used typemaps to return the value of the geometrical distance. This example does the exact opposite by returning the geometrical distance and using typemaps to return a reference to a structure. Why would we do this? (We’ve already done this in another way in the previous example).

This example demonstrates a down side of SWIG (current version) wrapper generation. We will observe if this feature can be implemented at all. So far, we have seen the different standard ‘C’ data types (such as ‘int’ and ‘double’) associated with INPUT and OUTPUT directives. We have also observed user defined data types (such as ‘point’), associated with the INPUT directive. This example examins the possibility of associating user defined data types with the OUTPUT directive.

To use this function in ‘Lua’ you update your case.i description file by adding the following declaration.

extern int return\_pointer\_distance(point \*INPUT, point \*INPUT, point \*OUTPUT);

This declaration should give you sufficient information about the ‘returns’ of the function call in ‘Lua’. Update your ‘load.lua’ file with the following script.

struct\_1 = case.point() -- Create a ‘point’ variable ‘struct\_1’

struct\_2 = case.point() -- Create a ‘point’ variable ‘struct\_2’

struct\_1.x = 1 -- Initialize the ‘point’ variable.

struct\_1.y = 2

struct\_2.x = 7 -- Initialize the ‘point’ variable.

struct\_2.y = 10

ans, st = case.return\_pointer\_distance(struct\_1, struct\_2)

print(ans)

**The above script will throw an error!**

“Error in return\_pointer\_distance expected 3..3 args, got 2”. Surprised?

Check on the declaration again:

extern int return\_pointer\_distance(point \*INPUT, point \*INPUT, point \*OUTPUT);

In this case, despite the fact that the declaration on the 3’rd argument of the function was marked OUTPUT, ‘Lua’ still expects a 3’rd argument. We can logically conclude that wrapping of the function for ‘Lua’ was incorrect.

Going beyond this stage for the sake of curiosity, let us fill the function call with 3 arguments instead of 2.

struct\_1 = case.point() -- Create a ‘point’ variable ‘struct\_1’

struct\_2 = case.point() -- Create a ‘point’ variable ‘struct\_2’

temp\_struct = case.point() –- Create a ‘point’ variable ‘temp\_struct’

struct\_1.x = 1 -- Initialize the ‘point’ variable.

struct\_1.y = 2

struct\_2.x = 7 -- Initialize the ‘point’ variable.

struct\_2.y = 10

ans, st = case.return\_pointer\_distance(struct\_1, struct\_2, temp\_struct)

print(ans) –- Will print ‘10’

print(st.x, st.y) –- Will throw error (‘st’ has a ‘nil’ value)

print(temp\_struct.x, temp\_struct.y) –- Will print 0 0

Since, ‘st’ has a ‘nil’ value, we can conclude that ‘Lua’ did not return 2 results like in the previous example.

**Workaround:** While using user defined data types, make sure to return them through functions but not using the OUTPUT typemaps.i directive.

#### void sum\_array(int arr[], int \*len, int \*res);

sum\_array() function calculates the sum of all the elements in the array. It accepts length as an input parameter, iterates over the length and returns the sum of the elements of the array through a reference‘res’.

void sum\_array(int arr[], int \*len, int \*res)

{

int i, sum = 0;

for(i = 1; i <= \*len; i++)

sum += \*(arr + i);

\*res = sum;

}

To test this function in ‘Lua’, update your case.i file with the following declaration.

extern void sum\_array(int arr[], int \*INPUT, int \*OUTPUT);

This declaration suggests that ‘arr’ is the source array (Which cannot be associated with OUTPUT or INPUT directives. We have seen this before) in the function which also accepts the length of the array as an input parameter and returns the sum of all the elements in the array through a reference.

To test out the function in ‘Lua’ add a script to the existing ‘load.lua’ file.

-- Declare ‘arr’ as integer array with size ‘5’ each

arr = case.new\_int\_array(5)

-- Call the function ‘sum\_array()’

print(case.sum\_array(arr, 5)

Output: 30

Observe the simplicity with which we use wrapped functions in the ‘Lua’ world. SWIG is indeed a marvelous tool.

#### Function pointers

In SWIG (current version), there is no direct method for dealing with function pointers. There is only an implicit technique that can be used but it has nothing to do with configuration in SWIG.

Function pointer returns are not supported by SWIG yet. Hence, we must write custom wrappers to manually perform this in an implicity manner. For example: foo(int (bar \*INOUT)(int)); is not supported. It is syntactically valid but does not work correctly. Below is an example for implicit usage of function pointers.

void init\_func\_pointer(void)

{

foobar = fact;

printf(“foobar(5): %d\n”, foobar(5));

}

In the case.i description file, add the following declaration.

extern void init\_func\_pointer(void);

Note that ‘fact()’ is a function for calculating factorial which is already defined in the beginning of this section. ‘foobar’ is a pointer to a function. It is declared as int (\*foobar)(int);

In ‘Lua’ case.init\_func\_pointer() will return ‘120’ for an output. This is one way of achieving results with function pointers in an implicit manner.

This section concludes the discussion of the different use cases that one might need to write scripts in ‘eLua’.

For additional information, check on the official SWIG manual that could be downloaded from the official website: [http://www.SWIG.org/doc.html](http://www.swig.org/doc.html).

# Lua tiny RAM (LTR)

LTR (Lua Tiny RAM) is a Lua patch for eLua that significantly decreases the RAM usage of Lua scripts, thus making it possible to run large Lua programs on systems with limited RAM. This section gives a full description of LTR. If you're writing eLua modules, this page will certainly be of interest to you, as it shows how to interact with LTR in a portable and easy to configure way.

The patch adds two new data types to Lua. Both or them are based on the lightuserdata type already found in Lua, and they share the same basic attributes: they don't need to be dynamically allocated (as they're just pointers on steroids) and they're compared in the same way lightuserdatas are compared (by value). And of course, they are not collectable, so the garbage collector won't have anything to do with them. The new types are:

1. **Lightfunctions**:

These are "simple" functions, in the sense that they can't have upvalues or environments. They are just pointers to regular C functions. Other than that, you can use them from Lua just as you'd use any other function.

1. **Rotables**:

These are read-only tables, but unlike the read-only tables that one can already implement in Lua with metamethods, they have a very specific property: they don't need any RAM at all. They are fully constant, so they can be read directly from ROM. They have a number of special features and limitations when compared with a regular table:

* 1. Rotables can only contain values of type "**lightfunction**", lua\_Number or pointers to other rotables.
  2. You can't add/delete/modify elements from rotables.
  3. You can use rotables as metatables for both "regular" tables and for Lua types.
  4. A rotable can have another rotable (or tself) as a metatable.
  5. You can iterate over rotables with pairs/ipairs/next just as you do with "regular" tables.

Just as with lightuserdata, you can only create lightfunctions and rotables from ‘C’ code, never from Lua itself.

## Importance of LTR

The entire code base, inclusive of functions and variables are all stored and maintained within the eLua state in memory. This would without doubt occupy significant amount of memory. LTR is a patch to address problem. LTR consideration becomes very significant especially in the embedded domain where memory is always limited. Consider the difference in memory consumption between the eLua system running with LTR optimizations and otherwise. Without LTR**,** the system indicated **27.14 KB** of memory consumption. When LTR was enabled, the memory consumption was **just 5.42 KB!** This is certainly a feature that must be considered and used in all embedded applications involving eLua.

## How to enable LTR

Enabling LTR is very easy: all you need to do is specify the optram=1 as a parameter to scons when building **eLua**, as explained. You don't even to specify this explicitly, as LTR is enabled by default for all eLua targets. When optram is 0, LTR is not active. In this mode the patch just tries to keep the modified version as close as possible to the unpatched version in terms of speed and functionality. You might want to use this if you want full Lua compatibility or need to overcome the read-only limitations of rotables. If your program behaves differently and you suspect that LTR might be the cause of your problems, recompiling with optram=0 is a quick way to eliminate or confirm your suspicions. When optram is 1 (default), all the LTR optimizations are enabled. The implementation of the Lua standard libraries is modified to take advantage of the new datatypes.

In particular, the IO library is modified to use the registry instead of environments, thus making it more resource-friendly, the side effect being that this mode doesn't support pipes in the **IO** module.

## LTR compatible SWIG

SWIG can normally be used for Lua wrapper generation. The wrappers can certainly run on micro-controllers but it consumes a lot of SRAM. The initial work for LTR compatible Lua language module for SWIG happened here at Infineon. Two new language modules have been introduced in SWIG: elua and eluac. They can be used for automatic LTR compatible wrapper code generation for eLua which could be used for testing and prototyping of new eLua modules. For now, the entire source tree is on sourceforge. It's an unofficial SWIG-2.0.4 release with support for eLua.

Sourceforge download link: [http://sourceforge.net/projects/eluaSWIG/files/](http://sourceforge.net/projects/eluaswig/files/)  
  
Working with elua and eluac is the same as working with the SWIG Lua module. No additional information is required. Read the following to understand how -elua is different from -eluac.

eLua-SWIG is a SWIG language extension for eLua. It generates wrappers compatible with the LTR patch supported by eLua.

This extension introduces two new SWIG switches: -elua and -eluac

-elua:

With this switch, the wrapper generated for eLua will support LTR. This extension puts all the C function wrappers and variable get/set wrappers in rotables. It also generates a metatable which will control how these variables are accessed from eLua.

Wrapping user defined datatype works differently. Wrapping of struct and union requires dynamic creation of metatables in order to manage the manipulation of such objects. Such functions cannot happen from within rotables.

Example:

struct point {   
int x, y;   
};

struct point p1;  
p1.x = 10;  
p1.y = 20;

In Lua: (When wrapped)

p1 = mod.point()  
p1.x = 10  
p1.y = 20

('mod' is the name of the module)

p1.x is an operation that is performed by a metamethod. Here, p1 gets its own metatable dynamically.

To incorporate this functionality, we must trade some amount of ram.

A tip: The higher the number of C functions that need to be wrapped, the more '-elua' will minimize ram consumption by putting them in rotables (With respect to the SWIG '-lua' language module). In other words, the difference between '-lua' and '-elua' (ram consumption) will be small for smaller modules.

If you don't care about this feature and want to save ram, use '-eluac'. (the 'c' in the switch is "crass compress")

'-eluac':

With this, no matter how huge the module, it will consume no ram. It puts all of the wrappers (function wrappers, structure variable get/set wrappers etc ...) in rotables. To access them from eLua, one must use functions.

Example:

To access the point structure above from eLua (wrapper is generated with '-eluac'):

mod.count\_set(0)

a = mod.new\_point()  
mod.point\_x\_set(a, 10)  
mod.point\_y\_set(a, 20)

mod.count\_set(1)

print(mod.point\_x\_get(a), mod.point\_y\_get(a), mod.count\_get())

'mod' is the module's name, 'count' is a name of a global variable in C, wrapped for eLua.

Please follow the instructions on the 'README' file to install eluaSWIG from source. This version of SWIG was installed and tested on the following system configuration: Linux 2.6.39-ARCH #1 SMP PREEMPT GNU/Linux. (Arch Linux distribution). Stellaris ARM Cortex-M3 luminary LM3S8962 was used as a standard board for all the wrapper code tests.

This version does not support compatibility with both optram = 0 and optram = 1 (It is not very hard to get this done though). If we don't use LTR, we can always use -lua with SWIG for eLua. Once you get eluaSWIG installed, invoke SWIG with -elua/c switch from your favourite terminal and supply the SWIG interface file (extension '.i'). The rotable we are interested in is 'SWIG\_commands[]'. By default, -elua or -eluac will put all the macro symbolic named constants (Eg: #define PI 3.14) and enum values in a rotable called 'SWIG\_constants[]'  
  
A sample wrapper file:

const LUA\_REG\_TYPE SWIG\_commands[] = {  
   {LSTRKEY("new\_int\_array"), LFUNCVAL(\_wrap\_new\_int\_array)},  
   {LSTRKEY("delete\_int\_array"), LFUNCVAL(\_wrap\_delete\_int\_array)},  
   {LSTRKEY("int\_array\_getitem"), LFUNCVAL(\_wrap\_int\_array\_getitem)},  
    ....  
};  
  
const LUA\_REG\_TYPE mt[] = {  
   {LSTRKEY("\_\_index"), LFUNCVAL(SWIG\_Lua\_module\_get)},  
   {LSTRKEY("\_\_newindex"), LFUNCVAL(SWIG\_Lua\_module\_set)},  
   {LSTRKEY(".get"), LROVAL(dot\_get)},  
   {LSTRKEY(".set"), LROVAL(dot\_set)},  
   {LNILKEY, LNILVAL}  
};  
  
const LUA\_REG\_TYPE SWIG\_constants[] = {  
   {LSTRKEY("PI"), LNUMVAL(3.14)},  
   {LSTRKEY("STRING"), LSTRVAL("Free as in freedom, GNU-GPL V3")},  
   {LSTRKEY("C"), LNUMVAL(C)},  
   {LSTRKEY("CPP"), LNUMVAL(CPP)},  
   ...  
};  
  
Note that mt[] is only generated with the -elua switch. -eluac puts everything in one rotable, SWIG\_commands. The wrapper introduces a new macro LSTRVAL, (just like LNUMVAL or LFUNCVAL) to handle macros like #define STR "Free as in freedom". To access such constants from eLua, we can write:   
print(mod.const.PI) -- Will print 3.14. ('mod' is the module name). Please read the eLua/c section in the TODO file in the source directory for known issues.

# Example programs for C-Lua interface with SWIG

So far, we discussed the intricacies of SWIG interfacing. Let us now move on to implementing what we have learnt. In this section, we will discuss one of the most important areas of data handling in the field of computation – Data structures. This manual does not serve as a beginner’s tutorial for data-structures. It simply addresses the easy implementation of linear data structures (As an example) in Lua by applying the power of wrapper generation in SWIG so as to extend the existing capabilities of eLua.

The Stack:

This data-structure works in a ‘Last in, first out’ configuration (Also known as ‘LIFO’). Given below is the header file declaration for the stack data-structure. File: stack,h

#ifndef STACK\_H

#define STACk\_H

// Dependancies

#include <malloc.h>

typedef struct stack {

int lim; // Limit the number of elements on stack

int top; // Represents the top of the stack

int \*box; // A reference to the stack with ‘lim’ elements

} stack;

void stack\_init(stack \*, int); // Initialize stack

int stack\_push(stack \*, int); // Push value into stack

int stack\_pop(stack \*); // Pop from stack

void stack\_print(stack \*); // Print the stack contents

#endif

The code for the above declaration is given below. In this example, the stack is implemented as an array.

File: stack.c

#include <stdio.h>

#include "stack.h"

void stack\_init(stack \*s, int num)

{

int i;

s->box = (int \*) malloc((sizeof(int) \* num));

for(i = 0; i < num; i++)

\*((s->box) + i) = 0;

s->top = -1;

s->lim = num;

}

int stack\_push(stack \*s, int num)

{

if(s->lim - 1 == s->top) {

printf("\nError: Stack overflow!\n");

return -1;

}

\*(s->box + (++s->top)) = num;

return s->top;

}

int stack\_pop(stack \*s)

{

if(s->top == -1) {

printf("\nError: Stack underflow!\n");

return -1;

}

int temp = \*(s->box + s->top);

\*(s->box + s->top) = 0;

--(s->top);

return temp;

}

void stack\_print(stack \*s)

{

int i;

for(i = 0; i < s->lim; i++)

printf("%d ", \*((s->box) + i));

}

To generate the wrapper code for the Lua interface, a ‘.i’ file must be written specifying the details of the functions and variables to be used in the Lua world.

Given below is a sample ‘.i’ file for the Stack program.

File: stack.i

#include <malloc.h>

%module ds

%include "typemaps.i"

%{

#include "stack.c"

%}

extern struct stack {

int lim;

int top;

int \*box;

};

typedef struct stack stack;

extern void stack\_init(stack \*, int);

extern int stack\_push(stack \*, int);

extern int stack\_pop(stack \*);

extern void stack\_print(stack \*);

extern void stack\_foo(const char \*OUTPUT);

Run the following on bash to create the stack\_wrap.c file which is linked with the interpreter.

raman@infineon:~/programs/stack$ SWIG –lua stack.i

This will generate a stack\_wrap.c file. This file needs to be included in the main interpreter ‘.c’ file, assuming we launch the interpreter session as a whole new process or linked with the main program through a static library.

A lua file needs to be written and loaded into the interpreter session. A sample lua code to perform stack operations can be written as follows.

File: stack.lua

stack = ds.stack()

ds.stack\_init(stack, 3)

ds.stack\_push(stack, 20)

ds.stack\_push(stack, 30)

ds.stack\_push(stack, 40)

ds.stack\_push(stack, 40)

ds.stack\_print(stack)

Output:

Error: Stack overflow!

20 30 40

The lua file is automatically loaded by the interpreter we built in the previous section. The output of the code sample is directly displayed on the terminal.

Stacks can also be implemented with a linked list. A linked list is a linear data structure which maintains links to other data elements called ‘nodes’. The idea of a node is based on the concept of a self referential structure.

Given below is an implementation of a doubly linked list (Two links) in ‘C’, with an interface to Lua via SWIG.

Doubly linked list:

File: dlist.h

#ifndef DLIST\_H

#define DLIST\_H

#include <stdio.h>

#include <malloc.h>

typedef struct node {

int info; // Information held within node

struct node \*next; // Link to the next node

struct node \*prev; // Link to the previous node

} node;

node \*dlist\_init(node \*); // Initialize linked list

node \*dlist\_newnode(node \*, int); // Create new node

int dlist\_search(node \*, int); // Search for an element

void dlist\_print(node \*); // Print linked list

#endif

The code for the above declaration is given below.

File: dlist.c

#include "dlist.h"

node \*dlist\_init(node \*head)

{

head = NULL;

return head;

}

node \*dlist\_newnode(node \*head, int info)

{

if(head == NULL) {

head = (node \*) malloc(sizeof(node));

(head)->info = info;

(head)->prev = NULL;

(head)->next = NULL;

return head;

}

node \*temp = head;

while(temp->next)

temp = temp->next;

temp->next = (node \*) malloc(sizeof(node));

temp->next->next = NULL;

temp->next->prev = temp;

temp->next->info = info;

return head;

}

int dlist\_search(node \*head, int inp)

{

int count = 0;

node \*temp = head;

while(temp) {

if(temp->info == inp) {

return count;

} else {

temp = temp->next;

++count;

}

}

return -1;

}

void dlist\_print(node \*head)

{

node \*temp = head;

while(temp) {

printf("%d ", temp->info);

temp = temp->next;

}

}

To generate the wrapper code for the Lua interface, another ‘.i’ file must be written specifying the details of the functions to be used in the Lua world.

Given below is a sample ‘.i’ file for the Dlist program.

File: dlist.i

%module ds

%{

#include "dlist.c"

%}

extern struct node {

int info;

struct node \*next;

struct node \*prev;

};

typedef struct node node;

extern node \*dlist\_init(node \*);

extern node \*dlist\_newnode(node \*, int);

extern int dlist\_search(node \*, int);

extern void dlist\_print(node \*);

Run the following on bash to create the dlist\_wrap.c file which is linked with the interpreter.

raman@infineon:~/programs/dlist$ swig –lua dlist.i

This will generate a dlist\_wrap.c file. This file needs to be included in the main interpreter ‘.c’ file, assuming we launch the interpreter session as a whole new process or linked with the main program through a static library.

A lua file needs to be written and loaded into the interpreter session. A sample lua code to perform linked list operations can be written as follows.

File: dlist.lua

head = ds.node()

head = ds.dlist\_init(head)

head = ds.dlist\_newnode(head, 10)

head = ds.dlist\_newnode(head, 20)

head = ds.dlist\_newnode(head, 30)

head = ds.dlist\_newnode(head, 40)

ds.dlist\_print(head)

print("Pos for search:", ds.dlist\_search(head, 20))

Output:

10 20 30 40 Pos for search: 1

Check on how the code magically works without the usage of typemaps! It is very important to understand when to use the feature of typemaps. (KISS principle: Keep it simple silly!)

Check on how the head pointer is represented in the above dlist example. With this representation, one might have to initialize ‘head’ on every call that manipulates the data structure (Like in the above example). To avoid this, the function must be declared so as to accept a double reference to ‘head’. This would work in the ‘C’ environment. In Lua, this can get messy since there is no absolute way to double reference a variable. For example, ds.node() in Lua calls the ‘C’ function which returns a pointer to the structure. It is impossible to pass a reference to this pointer to another function directly in Lua. One might have to manually write wrapper functions to support the usage of double pointers.

# Using SWIG with eLua for XMC4500

So, now we’ll take a look into a real example on how to use SWIG to wrap XMC4500 system libraries for their usage in eLua.

## IO002 DAVE3 App and its usage with eLua

Firstly, for using your IO002 DAVE3 App in eLua, you must write the SWIG interface file (.i file). For doing this, please first look into your DAVE3 generated code section of your DAVE3 project and look for your IO002 code there. By now, you must know that SWIG only reads an interface file (pretty much your header file in C) and generates your wrapper code for the eLua module. All you need to do now is to use your IO002\_Conf.h and IO002.h (from the DAVE3 generated code) in your SWIG interface file.

### Write your SWIG interface file

If you look into your inc/IO002 directory in your DAVE3 project, you should find the IO002.h and its corresponding configuration file IO002\_Conf.h after you have generated your code with DAVE3. The code should look pretty much like this:

IO002.h:

/\*\*

\* @file IO002.h

\*

\* Header file for IO\_Digital\_IO002 App

\*

\*/

**#ifndef** IO002\_H\_

**#define** IO002\_H\_

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* Include Files \*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**#include** <DAVE3.h>

/\*\*

\*This data type describes the App Handle

\*/

**typedef** **struct** IO002\_HandleType

{

/\*\* Port Number \*/

uint8\_t PortNr;

/\*\* Port pins \*/

uint8\_t PortPin;

/\*\* Port Regs \*/

PORTS\_TypeDef\* PortRegs;

}IO002\_HandleType;

/\*\*

\* This data type describes the Input Mode type

\*/

**typedef** **enum** IO002\_InputModeType

{

/\*\* Tri-state \*/

*IO002\_TRISTATE*,

/\*\* Input pull-down device connected \*/

*IO002\_PULL\_DOWN\_DEVICE*,

/\*\* Input pull-up device connected \*/

*IO002\_PULL\_UP\_DEVICE*,

/\*\* Pn\_OUTx continuously polls the input value \*/

*IO002\_CONT\_POLLING*,

/\*\* Inverted tri-state \*/

*IO002\_INV\_TRISTATE*,

/\*\* Inverted Input pull-down device connected \*/

*IO002\_INV\_PULL\_DOWN\_DEVICE*,

/\*\* Inverted Input pull-up device connected \*/

*IO002\_INV\_PULL\_UP\_DEVICE*,

/\*\* Inverted Pn\_OUTx continuously polls the input value \*/

*IO002\_INV\_CONT\_POLLING*,

}IO002\_InputModeType;

/\*\*

\* This data type describes the Output Mode type

\*/

**typedef** **enum** IO002\_OutputModeType

{

/\*\* Push pull output \*/

*IO002\_PUSHPULL* =0x10,

/\*\* Open drain output \*/

*IO002\_OPENDRAIN* =0x18

}IO002\_OutputModeType;

...

...

This above sample must be included in the SWIG interface file (Obviously, with SWIG specific macro identifiers)

IO002.i:

%module IO002 // Module name

%include "cpointer.i" // Support for pointers

%include "typemaps.i" // include typemaps

%{

#include <DAVE3.h>

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// these declarations will be directly included in

// your wrapper file IO002\_wrap.c

void IO002\_Init(void);

void IO002\_DisableOutputDriver(const IO002\_HandleType\* Handle,IO002\_InputModeType Mode);

void IO002\_EnableOutputDriver(const IO002\_HandleType\* Handle,IO002\_OutputModeType Mode);

%}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// the ‘rename’ identifier provides a way to access the

// wrapped functions with another name.

// With this, we can access the ‘IO002\_DisableOutputDriver’ function

// from eLua (on XMC) with the name ‘pin\_high’

%rename ("pin\_high") IO002\_DisableOutputDriver(const IO002\_HandleType\* Handle,IO002\_InputModeType Mode);

%rename ("pin\_low") IO002\_EnableOutputDriver(const IO002\_HandleType\* Handle,IO002\_OutputModeType Mode);

// With this, we can access the ‘IO002\_Handle0’ identifier

// from eLua (on XMC) with the name ‘Handle0’

%rename ("Handle0") IO002\_Handle0;

// Optionally, you could also do the same for enums.

// %rename (“OPENDRAIN”) IO002\_OPENDRAIN;

// With this, you should be able to access your enum like this:

// IO002.const.OPENDRAIN

// include this in your SWIG interface file to expose

// ‘IO002\_Handle0’ to eLua

extern const IO002\_HandleType IO002\_Handle0 = {

.PortNr = 3 , /\* Mapped Port \*/

.PortPin = 9, /\* Mapped Pin \*/

.PortRegs = (PORTS\_TypeDef\*)PORT3\_BASE /\* Port Base Address\*/

};

/\*\*

\*This data type describes the App Handle

\*/

struct IO002\_HandleType

{

/\*\* Port Number \*/

uint8\_t PortNr;

/\*\* Port pins \*/

uint8\_t PortPin;

/\*\* Port Regs \*/

PORTS\_TypeDef\* PortRegs;

};

typedef struct IO002\_HandleType IO002\_HandleType;

// enumerations are automatically wrapped by SWIG for eLua.

// All the enums can be accessed via the ‘const’ readonly

// table.

// for example, to access ‘IO002\_TRISTATE’ you could say

// something like this: IO002.const.IO002\_TRISTATE

/\*\*

\* This data type describes the Input Mode type

\*/

enum IO002\_InputModeType

{

/\*\* Tri-state \*/

IO002\_TRISTATE,

/\*\* Input pull-down device connected \*/

IO002\_PULL\_DOWN\_DEVICE,

/\*\* Input pull-up device connected \*/

IO002\_PULL\_UP\_DEVICE,

/\*\* Pn\_OUTx continuously polls the input value \*/

IO002\_CONT\_POLLING,

/\*\* Inverted tri-state \*/

IO002\_INV\_TRISTATE,

/\*\* Inverted Input pull-down device connected \*/

IO002\_INV\_PULL\_DOWN\_DEVICE,

/\*\* Inverted Input pull-up device connected \*/

IO002\_INV\_PULL\_UP\_DEVICE,

/\*\* Inverted Pn\_OUTx continuously polls the input value \*/

IO002\_INV\_CONT\_POLLING,

};

/\*\*

\* This data type describes the Output Mode type

\*/

enum IO002\_OutputModeType

{

/\*\* Push pull output \*/

IO002\_PUSHPULL =0x10,

/\*\* Open drain output \*/

IO002\_OPENDRAIN =0x18

};

// declare all your functions you want to use from eLua.

extern void IO002\_Init(void);

extern void IO002\_DisableOutputDriver(const IO002\_HandleType\* INPUT, IO002\_InputModeType);

extern void IO002\_EnableOutputDriver(const IO002\_HandleType\* INPUT,IO002\_OutputModeType);

### Generate your wrapper file with SWIG

This step is pretty simple once you have written your SWIG interface file.

raman@infineon:~/programs$ swig –lua –elua IO002.i

This command line invocation must have created your IO002\_wrap.c file in your working directory. (On windows, you could use the MinGW framework).

Now, you have created your wrapper file IO002\_wrap.c. Now, we could proceed towards including this wrapper file in eLua.

### Include your XMC module in eLua

1. Copy your wrapper file (IO002\_wrap.c) into this directory: **xLua2/xlua/src/modules**
2. **Optional**: Also copy your IO002.i into the same directory (for maintainability)
3. Goto directory **xLua2/xlua/src/platform/xmc4500**

Here, look for the file ‘exported\_apps.h’. This file includes all the references to include various XMC platform libraries in the eLua build for XMC. Your file should look something like this.

**#ifndef** EXPORTED\_APPS\_H\_

**#define** EXPORTED\_APPS\_H\_

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Include all DAVE3 Apps here.

**#define** EXPORTED\_APPS\_ROM\

\_ROM( AUXLIB\_IO002, luaopen\_IO002, IO002\_map )\

...

\_ROM( AUXLIB\_MODNAME, luaopen\_modname, modname\_map )

**#endif** /\* EXPORTED\_APPS\_H\_ \*/

**Note: ‘modname’ is the name of your XMC module which is included in your eLua build for XMC45.**

1. The next step to include the support for your module would be to add your new wrapped module in this file. So, you must define the \_ROM macro with the following entries in the ‘exported\_apps.h’ header file.

\_ROM( AUXLIB\_IO002, luaopen\_IO002, IO002\_map )

Your wrapped file (you just obtained by calling SWIG with the IO002.i file) contains references to the two parameters luaopen\_IO002 and IO002\_map. IO002\_map holds all the information (functions, pointers, variables, enumerations and basically all C code) that you wish to access from the eLua world on the XMC.

1. The first parameter AUXLIB\_IO002 (in the \_ROM macro) has to be defined by the user in the following file**: xLua2/xlua/src/modules/auxmods.h**

Your auxmods.h file should look like this:

**#define** AUXLIB\_ELUA "elua"

LUALIB\_API **int** ( **luaopen\_elua** )( lua\_State \*L );

**#define** AUXLIB\_I2C "i2c"

LUALIB\_API **int** ( **luaopen\_i2c** )( lua\_State \*L );

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// All DAVE3 Apps go here.

**#define** AUXLIB\_IO002 "IO002"

LUALIB\_API **int** ( **luaopen\_IO002** )(lua\_State \*L );

#define AUXLIB\_MODNAME “modname”

LUALIB\_API int ( luaopen\_modname )(lua\_State \*L );

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Helper macros

**#define** MOD\_CHECK\_ID( mod, id )\

**if**( !platform\_ **##** mod **##** \_exists( id ) )\

**return** luaL\_error( L, #mod" %d does not exist", ( **unsigned** )id )

**...**

You must basically define a function pointer called ‘luaopen\_modname’ in this file. This pointer is defined in the IO002\_wrap.c file.

1. Build your code base to access IO002 module from eLua

Now, you have pretty much all the prerequisites to build your XMC platform module. Now, you can build your DAVE3 project. Once, you have flashed your eLua program on the XMC, connect to your hyperterminal. You should see the eLua terminal. You could then fire off lua from the shell and start using your XMC platform module.

**Note: ‘modname’ is your XMC platform module name.**

### Using your XMC module in eLua

Now that you’ve already built your XMC platform module with SWIG, you should be able to access those modules from eLua (if you’ve build it correctly).

Let us now see how to use the IO002 module (we just built) from within Lua running on the XMC.

1. From the eLua shell, fire up ‘lua’ (You can also issues the ‘help’ command which gives you details on the supported commands).
2. You can access the IO002 specific functions through the IO002 table registerd in the Lua. Now, you could say something like this:

IO002.pin\_high(IO002.Handle0, IO002.const.IO002\_OPENDRAIN)

With this, your P3.9 pin (which you originally configured with DAVE3 in the IO002 App) should light up on your XMC board. You can also build your own scripts around these XMC modules and save them onto a memory card.

You can then invoke those scripts from the memory card through the eLua shell.

For example; You’d say something like this:

eLua# lua /mmc/IO002.lua

### Compiling your Lua scripts on your XMC

Now, you can optionally choose to compile your Lua scripts for the XMC device natively. The shell now provides a new option ‘luac’. This includes a modified version of the standard Lua compiler (which support various architectures and to choose between little and big endian) which can be used to compile your Lua scripts on your XMC board. Once you have compiled your Lua script, you can gain better performance and speed of your Lua script running on XMC. To do this, you can do the following:

1. From the shell, invoke the lua compiler like this:

luac /mmc/IO002.lua

This command should have created the default binary output file in your memory card called ‘luac.out’. You can also choose to change the default outuput name of the luac program by issuing the ‘-o’ flag followed by the name you wish to provide.

Once you have compiled the Lua script like this, you can execute the compiled binary output file from the shell like this:

lua /mmc/luac.out

This will directly execute your compiled file.

1. If you don’t care for the ‘debug’ option in Lua, you can also choose to strip the debug symbols. With this, you might save a few bytes on the file system. Stripping the symols while saving a few bytes will remove the possibility of using the compiled script with the Lua debug module.

You can strip the debug symbols like this:

luac –s /mmc/IO002.lua

You can also issue the ‘ls’ command (from the shell running on XMC) to check the size it occupies on the file system. You can see a difference of a significant number of bytes after you strip the symbols (when compared with the regular compiled script without stripping the symbols).

This concludes the example on how to use SWIG to wrap your XMC platform libraries and include the same in your eLua build. You can now access all your XMC platform libraries from within Lua (on XMC).

Please also note that the above procedure can be followed for wrapping any of the XMC platform libraries. The procedure is exactly the same.

# Appendix

## Building ‘Lua’ from source:

Download the source from the link given in the ‘Lua’ section and follow the given instructions.

1. raman@infineon:~/programs$ tar –zxvf lua-5-1-4

This will un-tar the tar ball and create a new directory with the source files.

1. ‘cd’ into the folder.
2. [raman@infineon:~/programs$](mailto:raman@infineon:~/programs/swig-2.0.3$) make linux

This will compile the source code and generate the executables. (This may take a while).

1. [raman@infineon:~/programs$](mailto:raman@infineon:~/programs/swig-2.0.3$) sudo make install
2. [raman@infineon:~/programs$](mailto:raman@infineon:~/programs/swig-2.0.3$) make clean (If needed).

This completes the build process for ‘Lua’.

You can now issue lua –v on the terminal to confirm successful installation. Just type lua on command line to enter into the interactive interpreter mode.

For troubleshooting/problems/bugs, check on the ‘Lua’ user group or the official ‘Lua’ documentation.

There is also a chat room for discussing Lua issues at [[irc.freenode.net #lua]](irc://irc.freenode.net/lua).

## Get eLua

To get eLua running on your development board, narrow down on the platform you will be using to install the tools necessary. This manual serves as an excellent guide for getting eLua up and running on your development board. Details about installing the tools on the Macintosh operating system will not be covered.

‘eLua’ is free software distributed in source code supported by the MIT license. ‘eLua’ source can be downloaded from the official ‘eLua’ website <http://www.eluaproject.net/en_downloads.html#binaries>. Binary versions for various processors are also available on the same link. Download the source code from the link and un-tar the source tar ball.

**Windows:**

The following instructions were tested on Windows 7 Professional, but they should work with little or no modification in any version of Windows 7, Vista or XP. You will have to download and install a few additional tools.

1. **Toolchain:** Download and install the Codesourcery toolchain. Codesourcery is free and open source. It is based on the GNU tool chain and is supported out of the box by ‘eLua’. You can download the tool chain from <http://www.codesourcery.com/sgpp/lite/arm/portal/subscription?@template=lite>. Be sure to download the EABI version on the link for windows.
2. ‘**Python’**: A good Windows distribution of Python is ActivePython. Simply download it and install it. Other Python distribution might work equally well. It can be downloaded from the main ActivePython site: <http://www.activestate.com/activepython>.
3. **‘Scons’**:This is a Python based replacement for the ubiquitous GNU based ‘make’. You can download ‘scons’ from <http://www.scons.org/download.php>.
4. **‘Lua’:** You must have already installed and tried ‘Lua’. In case you haven’t, refer to chapter 2 to understand how ‘Lua’ can be installed.
5. **‘MinGW’:** You require ‘gcc’ (GNU ‘C’ Compiler), an i368 toolchain to include a ROM file system in the final image of ‘eLua’. You can get it from <http://www.mingw.org/>. ‘Cygwin’ should work equally well.
6. **‘Teraterm’**: It is an SHH/Telnet client which supports terminal emulation and other communication protocols like XMODEM. (<http://www.ayera.com/teraterm/>). It is a substitute for ‘Hyperterminal’ on Windows platform. (Windows 7 does not include the Hyperterminal program by default). **‘Putty’** is also another alternative. (<http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html>).

**Note:**

All of the above tools work on the DOS console mode to generate the final ‘eLua’ image. Be sure to include these programs in the environment variable path. Make sure these programs can be called from command line. In most cases, with most programs, this is automatically done for you. Only in the case of ‘MinGW’ you might have to do this manually. To do this, perform the following.

1. Right click on ‘computer’ icon on start and hit properties on the menu.
2. Click on ‘Advanced system settings’ on the left pane.
3. Click on ‘Environment variables’
4. Add ‘C:\MinGW\bin’ (Or the MinGW folder you’ve chosen) to the ‘PATH’ variable.
5. Click Ok to exit the graphical menu.

**Gnu/Linux:**

The following instructions were tested on Ubuntu 11.04, 2.6.38-8-generic-pae #42 Ubuntu SMP i686 i368 GNU/Linux. It should however, work on the other versions of Ubuntu with little or no modification.

**Note:**

Ubuntu 11.04 was used to test some of the most interesting aspects of ‘eLua’. Features like remote procedure calling do not work on previous versions of Ubuntu. All the features of ‘eLua’ work on kernel 2.6.38 and above. (FDTI serial drivers are supported by kernel 2.6.35 and above).

Follow the instructions below.

1. **Toolchain:** Download and install the Codesourcery toolchain. You can download the tool chain from <http://www.codesourcery.com/sgpp/lite/arm/portal/subscription?@template=lite>. Be sure to download the EABI version on the link for Gnu/Linux.

By default, all downloaded ‘.bin’ files do not have ‘execute’ permission. Grant permission to execute the ‘.bin’ file.

raman@infineon:~/Downloads$ sudo chmod 777 arm-2011.03-42-arm-none-eabi.bin

raman@infineon:~/Downloads$ ./arm-2011.03-42-arm-none-eabi.bin

This will launch the GUI installer. Follow the instructions on the graphical console to proceed with the installation. This will complete the installation of the Codesourcery tool chain.

1. ‘**Python’:** A lot of application programs running on Ubuntu require Python. In most cases, Python will already be installed. You can confirm this by printing the version of Python on the terminal.

raman@infineon:~/Downloads$ python –V

Python 2.7.1+

If not, you will have to install Python manually.

raman@infineon:~/Downloads$ sudo apt-get install python

1. **‘Scons’:**

raman@infineon:~/Downloads$ sudo apt-get install scons

1. **‘Lua’:** You must have already installed and tried ‘Lua’. In case you haven’t, refer to chapter 2 to understand how ‘Lua’ can be installed.
2. **‘Gtkterm’:** It is a serial terminal emulator program for Gnu/Linux, specifically Ubuntu.

raman@infineon:~/Downloads$ sudo apt-get install gtkterm

Other options like ‘Putty’ and ‘Minicom’ will work equally well.

(<http://alioth.debian.org/projects/minicom>).

You are now set! (A lot of the above information can also be found online. <http://www.eluaproject.net/en_building_win.html>)

### Configuring the build image

**eLua** has a very flexible build system that can be used to select the components that are going to be part of the **eLua** binary image and also to set the compile time (static) configuration. To use it, you need to edit a single configuration file (platform\_conf.h) located in the platform specific directory (src/platform/<platform\_name>/platform\_conf.h).

The configuration parameters are described in detail below.

1. **Components:** An **eLua component** is a feature that can be enabled to add functionality to **eLua** itself, without modifying its API. An example of component configuration from platform\_conf.h is given below:

// Define here what components you want for this platform

#define BUILD\_XMODEM

#define BUILD\_SHELL

#define BUILD\_ROMFS

#define BUILD\_MMCFS

#define BUILD\_TERM

#define BUILD\_UIP

#define BUILD\_DHCPC

#define BUILD\_DNS

#define BUILD\_CON\_GENERIC

#define BUILD\_ADC

#define BUILD\_RPC

Any of these components may or may not be included in the final eLua image. Excluding a certain feature will save space but will remove the possibility of using that component.

1. **Modules:** You can also choose the modules that are going to be part of the **eLua** image. Unlike components, the modules have a direct impact on the **eLua** API, so choose them carefully. Disabling a module will save Flash space (and potentially RAM) but will also completely remove the possibility of using that module from **eLua**. The modules included in the build are specified by the **LUA\_PLATFORM\_LIBS\_ROM** macro. An example is given below:

#define LUA\_PLATFORM\_LIBS\_ROM\

\_ROM( AUXLIB\_PIO, luaopen\_pio, pio\_map )\

\_ROM( AUXLIB\_TMR, luaopen\_tmr, tmr\_map )\

\_ROM( AUXLIB\_PD, luaopen\_pd, pd\_map )\

\_ROM( AUXLIB\_UART, luaopen\_uart, uart\_map )\

\_ROM( AUXLIB\_TERM, luaopen\_term, term\_map )\

\_ROM( AUXLIB\_PWM, luaopen\_pwm, pwm\_map )\

\_ROM( AUXLIB\_PACK, luaopen\_pack, pack\_map )\

\_ROM( AUXLIB\_BIT, luaopen\_bit, bit\_map )\

\_ROM( AUXLIB\_CPU, luaopen\_cpu, cpu\_map )\

\_ROM( LUA\_MATHLIBNAME, luaopen\_math, math\_map )

Each module is defined by a **\_ROM( module\_name, module\_init\_function, module\_map\_array )** macro, where: **module\_name** is the name by which the module can be used from Lua, **module\_init\_function** is a function called by the Lua runtime when the module is initialized and **module\_map\_array** is a list of all the functions and constants exported by a module.

1. **Static configuration:** "Static configuration" refers to the compile-time configuration. Static configuration parameters are hard-coded in the firmware image and can’t be changed at run-time. These are macros that define specific aspects of each module in the micro-controller. For example, **CON\_UART\_ID, CON\_UART\_SPEED and CON\_TIMER\_ID** are all used to configure the eLua’s console I/O over serial interface.

For additional information, refer to the official eLua documentation online. (<http://www.eluaproject.net/en_building.html>)

### Invoking the build system

Once you have all the above tools in place, all one has to do is to invoke the build system (scons) with the right arguments. Scons supports a lot of build options. These options are used to fine tune the final target image of eLua. The different options for Scons are given below:

$ scons

[target=lua | lualong]

[cpu=<cpuname>]

[board=<boardname>]

[cpumode=arm | thumb]

[allocator = newlib | multiple | simple]

[toolchain = <toolchain name>]

[optram = 0 | 1]

[romfs = verbatim | compress | compile]

[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:

1. **cpu=name**: build for the specified CPU. A board name will be assigned by the build system automatically.
2. **board=name**: build for the specified board. The CPU name will be inferred by the build system automatically.
3. **cpu=name board=name**: build for the specified board and CPU. The build script won’t allow invalid CPU/board combinations.

For board/CPU assignment look at the beginning of the SConstruct file (the \_platform\_list), it’s self-explanatory. The other options are as follows:

1. **target=lua | lualong**: specify if you want to build "regular" Lua (with floating point support) or integer only Lua (lualong). The default is "lua". "lualong" runs faster on targets that don’t have a floating point co-processor, but it completely lacks support for floating point operations, it can only handle integers.
2. **cpumode=arm | thumb**: for ARM targets (not Cortex) this specifies the compilation mode. Its default value is thumb for AT91SAM7X targets and arm for STR9, LPC2888 and LPC2468 targets.
3. **allocator = newlib | multiple | simple**: choose between the default newlib allocator (newlib) which is an older version of dlmalloc, the multiple memory spaces allocator (multiple) which is a newer version of dlmalloc that can handle multiple memory spaces, and a very simple memory allocator (simple) that is slow and doesn’t handle fragmentation very well, but it requires very few resources (Flash/RAM). You should use the multiple allocator only if you need to support multiple memory spaces (for example boards that have external RAM). You should use simple only on very resource-constrained systems.
4. **toolchain=<toolchain name>**: this specifies the name of the toolchain used to build the image. In this case, “codesourcery”.
5. **optram=0 | 1**: enables of disables the LTR (Lua tiny ram) patch. The default is 1, which enables the LTR patch. (LTR will be covered in detail in later sections).
6. **prog**: by default, the above scons command will build only the elf (executable) 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).
7. **romfs = verbatim | compress | compile**: ROMFS compilation mode.
8. **boot = standard | luarpc**: Boot mode. standard will boot to either a shell or lua interactive prompt. luarpc boots with a waiting rpc server, using a UART and timer as specified in static configuration data.

The output will be a file named elua\_**[target]**\_**[cpu]**.elf (and also another file with the same name but ending in .bin/.hex if "prog" was specified for platforms that need these 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.

### Final build invocation sample

Open a terminal on your Gnu/Linux box (Or command prompt on your Windows box) and type in the following.

1. **‘cd’ into the elua source folder you downloaded.**
2. raman@infineon:~/Desktop/elua0.8$ scons cpu=LM3S8962 toolchain=codesourcery prog

This will generate two files, a ‘.elf’ file and a ‘.bin’ corresponding to your specific target which can directly burnt on the chip. If you use another CPU, specify its name instead of ‘LM3S8962’ keeping the other options.

### Flashing the eLua image

This is the final step to get eLua running on your micro-controller. The development board should usually come with additional drivers and flash tools on a compact disk. You will have to install them. This guide uses ‘LM3S8962’ as a prototype board. The same procedure is applicable to other development boards with different CPU’s.

1. **FTDI drivers: This driver is non-optional. This driver is used by the flash tool for COM port virtualization. These drivers will usually be provided on the compact disk. They can also be downloaded from the the official website. (**<http://www.ftdichip.com/Drivers/VCP.htm>**).**
2. **Flash tool: This tool can vary with development boards. The ‘Stellaris’ micro-controller uses the ‘LM-Flash programmer by Texas Instruments. Check your development board manul for specifications.**
3. **Keil: Keil is an optional tool. It can be used for the debugging features via the JTAG interface.**

**Launch the LM flash programmer. Select your CPU from the ‘manual configuration’ drop down menu. Give the path to the ‘.bin’ file (Or ‘.hex’ file) under the ‘program’ tab. Hit ‘program’.**

**This completes your eLua installation. You now have eLua running on your micro-controller! Unbelievable?**

**Let’s test it out.**

**Launch ‘Gtkterm’ (‘Putty’ or ‘Hyperterminal’ on Windows). Open a serial connction and set the following information.**

**Baud rate: 115200**

**Data bits: 8**

**Parity: NONE**

**Stop bits: 1**

**Flow control: NONE**

**After activating the connection, you can see the the ‘eLua’ prompt. Congratulations!**

**Note:**

Most advanced flash tools only work on Windows. If you’re using Gnu/Linux for a development environment, one could get Windows running on a virtual box for Gnu/Linux. If you’re brave, you can also try running the Flash tool with ‘Wine’ (<http://www.winehq.org/>).

## Writing an interface to the Lua interpreter for wrapper testing

Learning how to build an interface to the Lua interpreter is a requsite for understanding the extensible nature of Lua. A lot of samples included in this manual will involve the usage of this interpreter to test out different modules built. Writing an interface to the Lua interpreter is not a hard task. It only needs a working knowledge of the C-Lua API. The simplest of the Lua interpreter interfaces do not consume more than 30 lines of ‘C’ code!

By writing our own interface to the ‘Lua’ interpreter, we will be able to launch ‘Lua’ in a separate user process space which runs our custom Lua code, different from the space that runs the default ‘Lua’ interpreter, distributed with the source code. The source modifications we perform can be tested in this private process space.

This manual will not get into the depths of the C-Lua API. However, it will discuss the most important component of the C-Lua interaction – The Stack.

We face two problems when trying to exchange values between Lua and C: the mismatch between a dynamic and a static type system and the mismatch between automatic and manual memory management. It can be difficult to map such a complex type to other languages. Lua also does garbage collection: If we keep a Lua value in a C variable, the Lua engine has no way to know about this use; it may (wrongly) assume that this value is garbage and collect it.

Therefore, the Lua API does not define anything a fixed type, representing all Lua values. Instead, it uses an abstract stack to exchange values between Lua and C. Each slot in this stack can hold any Lua value. Whenever you want to ask for a value from Lua (such as the value of a global variable), you call Lua, which pushes the required value on the stack. Whenever you want to pass a value to Lua, you first push the value on the stack, and then you call Lua (which will pop the value). We still need a different function to push each C type on the stack and a different function to get each value from the stack, but we avoid the combinatorial explosion. Moreover, because this stack is managed by Lua, the garbage collector knows which values ‘C’ is using.

Nearly all functions in the API use the stack. The engine gets the function to be called from the stack and leaves any occasional error message there.

Lua manipulates this stack in a strict LIFO discipline (Last In, First Out; that is, always through the top). When you call Lua, it only changes the top part of the stack. Your C code has more freedom; specifically, it can inspect any element inside the stack and even insert and delete elements in any arbitrary position.

For more information on the C-Lua API, refer to the official Lua documentation. (<http://www.lua.org/pil/>)

Without further ado, let’s build a simple interface to the Lua interpreter. Check on the code segment given below.

Interpret.c

#include "lua.h"

#include "lualib.h"

#include "lauxlib.h"

#include <stdio.h>

static int lua\_interpret(lua\_State \*state, const char \*file)

{

state = luaL\_newstate(); // Open new Lua state.

luaL\_openlibs(state); // Flood the state with libraries.

// Load file and execute.

if(luaL\_loadfile(state, file) || lua\_pcall(state, 0, 0, 0))

return -1;

return 0;

}

int main(int argc, char \*\*argv)

{

lua\_State \*state;

if(lua\_interpret(state, "int.lua") == -1) {

printf("\nError: %s\n", lua\_tostring(state, -1));

return -1;

}

return 0;

}

As stated earlier, one can observe that the basic interface does not exceed 30 lines of ‘C’ code!

The source code is fairly self explainatory. (Read code comments). Be sure to include the sample Lua program ‘int.lua’ in the same directory as the ‘C’ interface. One can write different versions of the same program. It is also possible to interpret a line of Lua code per execution cycle. (This is exactly the inner working of the interactive Lua interpreter included in the source). While designing a library interface, it’s easier to organize sample Lua code segments in files and interpret them as a whole.

For additional information, look into the Lua source code.

int.lua (Sample)

print(“Hello world!”)

print(30 + 20)

Output:

Hello world!

50

Save the above program in a directory on your file-system. It will certainly come to use when dealing with the wrapper generation with SWIG.

**Note:**

For the above program to work, it is important to have the <lua.h>, <lualib.h> and <lauxlib.h> placed in appropriate places in your file system. The easiest way to achieve this would be to install ‘Lua’ from the source. If you install the binary package, you will no doubt be able to execute ‘Lua’ programs but you will be unable to play around with the ‘Lua-C’ API. If you haven’t installed ‘Lua’ from the source tar-ball, download the tarball from the official ‘Lua’ website and follow the instructions for installation from source, given in this section.