The SWIGLIB project*

Luigi Scarso

Abstract

The SWIGLIB project aims to show a way to build and distribute shared libraries for LuaTEX by means of SWIG. This paper depicts the infrastructure that has been created and the rationale behind it. Simple examples are shown.

1 Introduction

The Lua language is well-known for its simplicity and compactness, and also for its easy integration into an existing project. This integration refers both to compilation — TEX Live currently provides binaries for 21 platforms and all of them have a `luatex` executable — and in a more general sense the relatively small amount of time required to get acquainted with its constructs and data structures.

Analogous to the `\usepackage` macro of LATEX, it is easy to extend the built-in features of Lua by means of external Lua modules, usually loaded with `load("⟨module_name⟩")`. What perhaps is less well known is that the same is also available for *binary* modules; for example, a C library compiled in the native format of the platform. This is due to the double nature of Lua, as both an interpreted language and a library that can be linked with an application (see [4, p. 249]):[1] the interaction of the Lua library and the application must follow the application programming interface (API) of Lua.

While for LuaTEX there is currently no official C API — it's a program, not a library — the Lua API is completely described in the Lua manual (`http://www.lua.org/manual`) and it counts 245 items, including constants, macros, functions and standard libraries. They consistently use a stack to exchange data (and hence several functions are dedicated to the stack manipulation) and use an opaque data structure to store the current state, but the stack is accessible only by the state and sometimes it is confused with it. By design (related to the choice of ANSI C for the implementation language) the Lua state is not thread-safe, but the library is carefully designed to avoid destructive interference in global variables and in some cases multithreading with a single shared state appears to be possible [13]. In any case, the best solution is to avoid sharing the state between multiple threads — the library can in fact safely manage different states, at the price of more complex code.

```
-- a Lua function that adds two numbers
function add ( x, y )
  return x + y

/* The C code that calls the Lua function; */
/* we suppose that the state L             */
/* is already initialised.                 */

/* Lua headers */
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>

int lua_add ( int x, int y ){
 int sum;
 lua_getglobal(L, "add");  /* function name   */
 lua_pushnumber(L, x);     /* first argument  */
 lua_pushnumber(L, y);     /* second argument */
 lua_call(L, 2, 1);        /* call the function
            with 2 arguments, return 1 result */
 sum = (int)lua_tointeger(L, -1);/* get result*/
 lua_pop(L, 1);            /* clear the stack */
 return sum;               /* return the sum  */
}
```

Figure 1: Calling a Lua function from C.

Every "well done" C library exposes its services by means of an API which is, of course, completely unrelated to the Lua API. Communication between the two can happen in either direction: when the application library wants to execute a Lua function it has to follow the Lua API as shown for example in fig. 1, and similarly when a C function is called by the Lua interpreter (see fig. 2) — and this latter case is the subject of this paper. It's clear that if an application library has tens or hundreds of functions, writing the corresponding code can take a considerable amount of time.

Before discussing the tools and the infrastructure used, it's worth mentioning at least these three scenarios where an application library can be useful:

- *pre/post processing* of data, typically pre-processing images (i.e. conversion) and post-processing PDFs;
- *extending* LuaTEX, for example to connect to a database at runtime;
- *extending* the application with LuaTEX as a scripting language — probably a less common, but still important, use.

* In fulfillment of the TEX Development Fund grant no. 23, *Dynamic library support in LuaTEX*, 2013. Grants from (in alphabetical order) CSTUG, DANTE e.V., GUST, NTG and TUG.

[1] By design the standard Lua library is written in ANSI C and it is precisely for this reason that integration into disparate platforms is easy.

```
/* Example C function to be called from Lua. */

/* Lua headers  */
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>

int c_add (int x, int y) {
 return x+y;
}

int _wrap_c_add (lua_State *L) {
 int x,y, sum;
 x = (int)lua_tointeger(L, -1); /* first arg   */
 y = (int)lua_tointeger(L, -2); /* second arg  */
 sum = c_add(x,y);              /* call c_add  */
 lua_pushnumber(L, sum);        /* push result */
 return 1;                      /* return sum  */
}

static const luaL_Reg myapplication [] = {
  {"add", _wrap_c_add},  /* register c_add */
  {NULL,NULL}            /* sentinel       */
};

int luaopen_myapplication(lua_State *L) {
 luaL_newlib(L,myapplication);
 return 1;
}

-- Calling c_add from Lua
local myapplication = require("myapplication")
print (myapplication.add(2,3))
```

Figure 2: Calling a C function from Lua.

2 The SWIG tool

As described above, to connect an application library with the Lua interpreter a third layer which acts as interface is needed. This layer, called *wrapper code*, must know the application API and, of course, the Lua API. In fig. 2, `c_add` is the application function, and the wrapper code items are `_wrap_c_add`, `myapplication` and `luaopen_myapplication`; the local Lua variable `myapplication` is the *binding*. Under Linux the compilation is straightforward:

```
$ gcc -I/usr/include/lua5.2 -fPIC \
  -o myapplication.o \
  -c myapplication.c
$ gcc -I/usr/include/lua5.2 -shared \
  -o myapplication.so myapplication.o \
  -llua5.2
```

where `-fPIC` tells the compiler to generate position independent code, given that `myapplication.so` is a shared library. From this elementary example we can identify the following issues:

- how to generate a wrapper for a rich and complex application API?
- how to compile the wrapper to obtain a suitable binary module?
- how to distribute the module?

The next subsections will try to address these questions.

2.1 Generate a wrapper

After a initial period of experimentation the following assumptions have emerged as suitable for a project that aims to serve the TEX community:

1. the wrapper code should be generated in an automatic fashion preserving as much as possible the meaning and the names of the functions and data structures of the application API;
2. the application and Lua API should be freely accessible.

The tool chosen is SWIG, the *Simplified Wrapper and Interface Generator* program available for different platforms, including Linux, Windows and Mac OS X. Its web site is `http://www.swig.org`; for a quick overview, see also `http://www.ibm.com/developerworks/aix/library/au-swig`. SWIG has a powerful C/C++ preprocessor and can analyse[2] a header file and produce the wrapper code. For example, given the C API

```
/* myapplication.h */
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>

extern int c_add (int, int);
```

the SWIG interface file to create the wrapper is:

```
%module core
%{
/* code included in the wrapper */
#include "myapplication.h"
%}
/* header to analyse */
%include "myapplication.h"
```

The wrapper itself (by default `core_wrap.c`) is generated with

```
$ swig -lua core.i
```

and, supposing that the application header and the shared library `myapplication.so` live in the current

[2] SWIG works particularly well with C libraries, while with C++ libraries usually the developer has to manually write some customisation, e.g. to manage function overloading or multiple inheritance. For C++, in fact, "at the lowest level, SWIG generates a collection of procedural ANSI C-style wrappers"; see `http://www.swig.org/Doc3.0/SWIGDocumentation.html#SWIGPlus_nn2`.

directory `./`, the binary module `core.so` is compiled as below (again for the Linux platform):

```
$ gcc -I/usr/include/lua5.2 -I./ -fPIC \
      -o core_wrap.o \
       -c core_wrap.c
$ gcc -L./ -Wl,-rpath,'$ORIGIN/.' -shared \
      -o core.so core_wrap.o \
      -lmyapplication -llua5.2
```

and loaded in Lua with

```
local myapplication = require("core")
print (myapplication.c_add(2, 3))
```

This example shows all the basic components used in the SWIGLIB project. A practical interface file is in fact only a bit more complex: here is one for the cURL library, a free and easy-to-use client-side URL transfer library (`http://curl.haxx.se/libcurl`): [3]

```
%module core
#ifdef SWIGLIB_WINDOWS
%include <windows.i>
#endif

/* Section for utilities, such as  */
/* built-in wrappers for C arrays, */
/* C pointers, function pointers.  */
...
*/

/* API */
%{
#include "curl/curl.h"
%}

/* Headers to generate the wrapper */
%include "curl/curlver.h"
%include "curl/curlbuild.h"
%include "curl/curlrules.h"
%include "curl/curl.h"
%include "curl/easy.h"
%include "curl/multi.h"

/* Customisation   */
%include "native.i"
%include "inline.i"
%include "luacode.i"
```

Each binary module of the SWIGLIB project is named `core`, so each needs to be saved into a specific directory, as will be shown later. Next, there is a section to eventually include the wrappers that SWIG supplies by default for the basic C types such as `char`, `int`, `long`, etc. (useful, for example, when a parameter of a function is an array or a pointer to a basic type). After that is the section that includes the application API into the wrapper and generates the wrapper; the order of the `%include` directives is not random, but reflects the dependencies between the headers.[4] Finally, the `native.i` file is used when the developer wants to replace the standard SWIG wrapper of a function with a custom implementation; the `inline.i` file is useful to add new members to the application API; and the `luacode.i` file to add Lua code when the module is initialised at loading time.

Normally, these `.i` files are empty but it turns out that our example of the cURL API has several functions that take a variably-typed argument — either a pointer to a long or a pointer to a char, etc.; in any case, a finite set of types as described in the documentation of the API. Here the `inline.i` file defines, for each variation of such functions, several C helper functions with the third argument fixed; i.e. one function with a pointer to a long, a second with a pointer to a char, etc. The `luacode.i` file has the single Lua function that calls the helper functions with the right third argument: of course this means that a lot of code is hand-written, given that a single function can have 3 or 4 helper functions — it sounds complicated but it's not especially difficult.[5]

In most cases this simple organisation of the interface file is enough, but it can be extended in two ways: first, to build a *helper* module that consists solely of SWIG directives as in

```
%module core
#ifdef SWIGLIB_WINDOWS
%include <windows.i>
#endif
%include "carrays.i"
%include "cpointer.i"
%include "constraints.i"
%include "cmalloc.i"
%include "lua_fnptr.i"

%{ /* array functions */ %}
%array_functions(char, char_array);
%array_functions(unsigned char, u_char_array);
%array_functions(char*, char_p_array);
%array_functions(unsigned char*, u_char_p_array);
/* Several other SWIG directives ...*/
```

Second, by adding C functions and data structures to the inline interface a user can build a custom *usermodule*, eventually using other application libraries. In other words, SWIG also supports interface files `usercore.i`, `usernative.i`, `userinline.i` and

[3] The real file has a few more directives, but this example shows the important pieces.

[4] `gcc -H` can be used with a header file to print out its dependencies.

[5] Although the chapter "Variable Length Arguments" at `http://www.swig.org/Doc3.0/SWIGDocumentation.html#Varargs` does start with *a.k.a. "The horror. The horror."*

`userluacode.i` and hence a `usercore` binary module that stays in the same directory as the `core` application.

2.2 Compilation of a wrapper

Compilation of binary modules is not as difficult as it seems at first sight: given an application header and the *corresponding* shared library, SWIG generates ANSI C wrapper code, which is usually both portable and easily compilable. Of course much depends on the application library, but currently all the modules provided are compiled for 64-bit Linux (Ubuntu 14.04 LTS) with the GCC toolchain and cross-compiled for Microsoft Windows 32-bit and 64-bit using the Mingw-w64 toolchain; it's also possible under Linux to use the native compiler suite for Windows from `http://tdm-gcc.tdragon.net`

In this way the application headers and library match among different platforms (only two in this phase) which in turn means that at the LuaTEX level the interface to the application library is the same. While the compilation of an application module almost always uses the `configure` script generated from the GNU Autotools, SWIGLIB uses for the wrapper simple `bash` scripts; for example, for `curl` under Linux:

```
trap 'echo "Error on building library";
      exit $?' ERR
echo "building for      : linux 64bit"
## SWIG
swig -I$(pwd)/resources/include64  -lua \
      -o core_wrap.c ../core.i
## Compile wrapper
rm -f  core_wrap.o
gcc -O3 -fpic -pthread -I$LUAINC \
     -I./resources/include64/  \
     -c core_wrap.c  \
     -o core_wrap.o
## Build library
rm -f core.so
CFLAGS="-g -O3 -Wall   -shared  \
        -I./resources/include64 \
        -L./resources/lib64"
LIBS="-lcurl -lssh2"
gcc $CFLAGS  -Wl,-rpath,'$ORIGIN/.' \
    core_wrap.o                     \
    $LIBS                           \
    -o core.so
## End
mv core.so resources/lib64
rm core_wrap.o
rm core_wrap.c
```

and for Windows 64-bit it's almost the same:

```
trap 'echo "Error on building library";
      exit $?' ERR
## SWIG
swig -DSWIGLIB_WINDOWS \
     -I$(pwd)/resources/include64 \
     -lua -o core_wrap.c ../core.i
## Compile the wrapper
rm -f  core_wrap.o
$GCCMINGW64 -O3 -I$LUAINC  \
    -I./resources/include64/ \
    -c core_wrap.c        \
    -o core_wrap.o
## Build the library
rm -f core.dll
CFLAGS="-O3 -Wall -shared "
LIBS="$LUALIB/$LUADLL64
resources/lib64/libssh2-1.dll
resources/lib64/zlib1.dll
resources/lib64/libcurl-4.dll
resources/lib64/ssleay32.dll
resources/lib64/libeay32.dll "
$GCCMINGW64 $CFLAGS  \
    -Wl,-rpath,'$ORIGIN/.' \
    core_wrap.o            \
    $LIBS                  \
    -o core.dll
## End
mv core.dll resources/lib64
rm core_wrap.o
rm core_wrap.c
```

A simple bash script should be easily ported to different platforms: the GNU Autotools are well suited for Unix-like systems, but Windows has its own toolchain and such a a shell script can be translated in a batch script without much effort, giving a good starting point (see [1, p. 3]).

A binary module can easily depend on other binary modules. Under Windows, these modules are searched first in the same directory of the wrapper, but in Linux (and hopefully on other Unix-like systems too) that "local" search is enforced with the linker option `-Wl,-rpath,'$ORIGIN/.'`. We do this to keep the wrapper module and its dependencies as much as possible self-contained in a TDS tree.

In spite of the efforts to mask the differences between the systems, at some point they emerge and it's not always possible to find a nice way to manage them. One of these differences is symbol resolution and collision: when an application module has a reference to an external symbol (i.e. a function or a data item), under Linux this reference is resolved at run-time while in Windows it must be resolved at build-time, when the module is compiled.

Given that an application module always needs to resolve the Lua API symbols, the first consequence is that the `luatex` Windows binary must be compiled with a dynamic link to an external Lua library (a Lua DLL) and the same DLL must be used at build-time for the application module. Under Linux the Lua

API symbols are unequivocally resolved inside the `luatex` binary, but if the application module needs a symbol from another API (for example, a function from `libpng`, which is part of `luatex`) it must resolve that symbol to an external auxiliary library and not inside the `luatex` binary: with Windows this happens automatically because, by default, the symbols are not visible if not explicitly marked as such, but in Linux the situation is just the opposite. The `luatex` binary must be compiled with the `gcc` flag `-fvisibility=hidden` — this will be the default starting with TEX Live 2015:[6] hence, all the Linux binaries before this date are not safe.

Another fundamental difference is that Linux 64-bit and Windows 64-bit don't use the same data model. Linux uses the so-called LP64, where the type `long` and a pointer are both 64 bits, while Windows uses LLP64, where a `long` is 32 bits and a pointer 64 bits. As a consequence, if a program under 64-bit Linux uses a `long` to store an address, it cannot be automatically ported to 64-bit Windows. Although the 64-bit Windows version could use the type `long long`, this is a a C99 extension and it's not supported by the Microsoft Visual C compiler. The situation is no better in C++: the following example that uses GMP 6.0.0 fails to compile with Mingw-w64 but works with GCC under Linux[7] — and in both cases `sizeof a` returns 8.

```
#include <gmpxx.h>
#include <iostream>
using namespace std;
int main(void) {
    size_t a = 5;
    mpz_class b(a);
    cout << b.get_ui() << endl;
    cout<< sizeof a <<endl;
    return 0;
}
```

3 Deployment

The SWIGLIB project is hosted[8] at `http://swiglib.foundry.supelec.fr` with a public readonly Subversion source repository accessible at `http://foundry.supelec.fr/projects/swiglib`. The root has currently the following application modules:

```
trunk
├ attic
├ basement
├ curl
├ experimental
├ ghostscript
├ graphicsmagick
├ helpers
├ leptonica
├ libffi
├ lua
├ luarepl
├ mysql
├ parigp
├ physicsfs
├ postgresql
├ qpdf
├ R
├ sqlite
├ swig
├ usermod
└ zeromq
COPYRIGHT
build.sh
```

Each application module has the following layout (here shown for `curl`):

```
curl
└ 7.40.0
    ├ docs
    ├ linux
    │  ├ resources
    │  │  ├ include32
    │  │  ├ include64
    │  │  │  └ curl
    │  │  ├ lib32
    │  │  └ lib64
    │  └ test
    │  build-linux-x86_64.sh
    ├ osx
    │  └ resources
    │      ├ include32
    │      ├ include64
    │      ├ lib32
    │      └ lib64
    │
    └ windows
        ├ resources
        │  ├ include32
        │  │  └ curl
        │  ├ include64
        │  │  └ curl
        │  ├ lib32
        │  └ lib64
        └ test
        build-mingw32.sh
        build-mingw64.sh
        build-msys32.sh
        build-msys64.sh
    core.i
    inline.i
```

[6] Peter Breitenlohner has done incalculable work in implementing the symbol visibility and the build of shared versions of the TEX-specific libraries.

[7] And the fork MPIR 2.7.9 compiles correctly under Mingw-w64 and gives the same result as Linux!

[8] Thanks to Fabrice Popineau for his invaluable support.

```
luacode.i
native.i
```

where `lib64` (`lib32`) hosts the application API and `lib64` (`lib32`) the binary module. The `osx` directory is a placeholder — currently it's empty. The `lua` directory contains the Lua API and the binaries for Linux 64-bit, Windows 32-bit and 64-bit:

```
└─ luatex-beta-0.79.3.1
    ├─ include
    │   ├─ lauxlib.h
    │   ├─ luaconf.h
    │   ├─ lua.h
    │   ├─ lua.hpp
    │   ├─ lualib.h
    │   ├─ patch-01-utf-8
    │   ├─ patch-02-FreeBSD
    │   └─ patch-03-export
    ├─ linux
    │   └─ luatex
    ├─ w32
    │   ├─ libkpathsea-6.dll
    │   ├─ luatex.exe
    │   └─ texlua52.dll
    └─ w64
        ├─ libkpathsea-6.dll
        ├─ luatex.exe
        └─ texlua52.dll
```

3.1 Application module location in the TDS

The natural location of a binary module inside a TDS directory is under `bin/`. The current layout looks like the following (for Linux 64-bit):

```
tex
└─ texmf-linux-64
    └─ bin
        └─ lib
            └─ luatex
                └─ lua
                    └─ swiglib
                        └─ curl
                            └─ 7.40.0
                                ├─ core.so
                                └─ libcurl.so
```

SWIGLIB doesn't require a particular method to load a wrapper module, because this is a task of the format. The tests in the Subversion repository use the low-level Lua function `load`, but they need to know the system and the full path of the module; on the other hand, ConTEXt has a global `swiglib` function (see `util-lib.lua` and [3]) that is independent from the system and the path — but it doesn't use the `kpse` library.

4 Conclusions

Without a doubt, building a wrapper module requires a working knowledge at least of the C language, for which [5] is still a pleasure to read; useful information on shared libraries is also in [2] and [7] while for Linux [6] is still one of the best references, as [10] and [9] are for Windows. Moreover, having a working wrapper is only half of the story: the rest is a working Lua/TEX layer that suits with the format in use — and this cannot be part of the underlying SWIGLIB. The example with GMP 6.0.0 shows that an application module that compiles well and passes all the tests can still fail to compile an apparently innocuous program. The C code itself is not always easy to understand, as for example with the following program

```
/* test.c    */
#include <stdlib.h>
void foo(int *x){
  int y = *x;
  if (x == NULL)
    return;
  return;
}
int main(){
  int *x;
  x = NULL;
  foo(x);
  return 0;
}
```

which gives a segmentation fault if compiled with `gcc` without optimisation (`gcc -o test test.c`), but it's ok with optimisation (`gcc -O3 -o test test.c`).[9] Portable multithreading also looks problematic, due to the lack of support in ANSI C and hence in Lua. Of course the Linux and Windows platforms are not the only ones to consider and the absence of Mac OS X is the most notable; FreeBSD as well, which seems to be rather easier to add.

Despite these issues, SWIG is an exceptionally flexible program, and it can adapted to manage almost any situations. If an interface file is complicated, it can often be simplified with an auxiliary C module; if a user needs to customise an application module, this can be done by adding a set of Lua functions and/or C functions — and all this while always formally writing an interface file. A possible objection is that LuaTEX does not have a read-eval-print loop ("repl") program as standard Lua does, but SWIGLIB has a pure Lua module `luarepl` that mimics the original one quite well. This means that it's possible to use

[9] `y = *x` results in undefined behaviour when `x` is `NULL`, but the optimisation `-O3` is able to detect that `y` is never used and it deletes it.

LuaTEX as a general-purpose scripting language, i.e. to manage the installation of TEX packages.

Regarding LuaJITTEX [12]: even when it's possible to use the same interface file, the API and the `luajitex` libraries are not the same. Furthermore, LuaJIT users seem to prefer the use of the LuaJIT `ffi` module, which is roughly similar to SWIG. It should still be doable to implement in SWIG via a new backend LuaJIT-ffi that emits `ffi` chunks instead of the LuaJIT C API, effectively eliminating the need for a C compiler. Clearly work for the future.

Some practical examples of applications are shown in [3] and [11]. These will also be the subject of a future paper.

References

[1] John Calcote. *Autotools: A Practitioner's Guide to GNU Autoconf, Automake, and Libtool*. No Starch Press, San Francisco, CA, USA, 1st edition, 2010.

[2] Ulrich Drepper. How to write shared libraries. `http://www.akkadia.org/drepper/dsohowto.pdf`, December 10 2011. Accessed: 2015-03-5.

[3] Hans Hagen. Swiglib basics. `http://minimals.contextgarden.net/current/doc/context/pragma/general/manuals/swiglib-mkiv.pdf`.

[4] Roberto Ierusalimschy. *Programming in Lua, Third Edition*. Lua.Org, 3rd edition, 2013.

[5] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[6] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, San Francisco, CA, USA, 1st edition, 2010.

[7] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1999.

[8] T. Przechlewski, editor. *What Can Typography Gain from Electronic Media?* Polska Grupa Użytkowników Systemu TEX — GUST, 2014. ISBN 9788393901609. `http://books.google.it/books?id=abDNoAEACAAJ`.

[9] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals, Part 1: Covering Windows Server 2008 R2 and Windows 7*. Microsoft Press, 6th edition, 2012.

[10] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals, Part 2: Covering Windows Server 2008 R2 and Windows 7 (Windows Internals)*. Microsoft Press, 2012.

[11] Luigi Scarso. Extending ConTEXt MkIV with PARI/GP. *ArsTEXnica*, 11:65–74, April 2011. `http://www.guitex.org/home/images/ArsTeXnica/AT011/AT11-scarso.pdf`.

[12] Luigi Scarso. LuaJITTEX. *TUGboat*, 34(1):64–71, 2013. `http://tug.org/TUGboat/34-1/tb106scarso.pdf`.

[13] Luigi Scarso. Some experiments with OpenMP and LuaTEX. In Przechlewski [8]. `http://www.gust.org.pl/bachotex/2014-pl/presentations/openmp-slides.pdf`.

◇ Luigi Scarso
`luigi dot scarso (at) gmail dot com`
`http://swiglib.foundry.supelec.fr`