Lua in the NetBSD Kernel

Marc Balmer <mbalmer@NetBSD.org>
NetBSD, in general, is meant as a „stable research platform” – that is, a system that can be used for commercial, home, and research work. . . what **you** do with it is up to you. In general, those of us working on NetBSD are trying to improve the system in whatever way we can – support for more hardware, stability, performance, documentation. . .

*Chris G. Demetriou*
Topics

1. **The Programming Language Lua**
   - The Lua Interpreter
   - Syntax and such
   - Modules

2. **Embedding Lua in C Programs**
   - State Manipulation
   - Calling C from Lua
   - Calling Lua from C

3. **Lua in the NetBSD Kernel**
   - Use Cases
   - Implementation Overview
   - Implementation Details
Builds in all platforms with an **ANSI/ISO C** compiler
Fits into **128K ROM, 64K RAM** per interpreter state\(^1\)
**Fastest** in the realm of interpreted languages
Well-documented **C/C++ API** to extend applications
One of the fastest mechanisms for **call-out to C**
Incremental **low-latency garbage collector**
**Sandboxing** for restricted access to resources
**Meta-mechanisms** for language extensions, e.g. class-based **object orientation** and inheritance
**Natural datatype** can be integer, float or double
Supports **closures** and cooperative **threads**
Open source under the **OSI-certified** MIT license

\(^1\) Complete Lua SOC, practical applications in 256K ROM / 64K RAM

Designed, implemented and maintained at the Pontifical Catholic University of Rio de Janeiro  [www.lua.org](http://www.lua.org)
Running Lua Sourcecode
Compiling / Running Lua Bytecode

Lua Source → Compiler → Bytecode → Runtime

```
local n
n=n + 1
print(n)
```
The Lua Interpreter

Running from C

- int luaL_dofile(lua_State *L, const char *filename)
- int luaL_dostring(lua_State *L, const char *str)
Values, Variables, and, Data Types

- Variables have no type
- Values do have a type
- Functions are first-class values
Tables

- Tables are THE data structure in Lua
- Nice constructor syntax
- Tables make Lua a good DDL
- Metatables can be associated with every object
Syntax and such

Lua Table Constructor

Create and initialize a table, access a field:

mytable = {
    name = 'Marc',
    surname = 'Balmer',
    email = 'm@x.org'
}

print(mytable.email)
Extending Lua Programs

Acess GPIO pins from Lua:

```lua
require 'gpio'

g = gpio.open('/dev/gpio0')
g:write(4, gpio.PIN_HIGH)
g:close()
```
Creating and Destroying a State

- `lua_State *L = lua_newstate()`
- `luaopen_module(L)`
- `lua_close(L)`
Calling a C Function

• Function has been registered in luaopen_module()
• int function(lua_State *L)
• Parameters popped from the stack
• Return values pushed on the stack
• Return Nr. of return values
Calling a Lua Function

- Find the function and make sure it *is* a function
- Push parameters on the stack
- Use `lua_call(lua_State *L, int index)`
- or `lua_pcall(lua_State *L, int index)`
- Pop return values from the stack
Calling a Lua Function

The Lua function

function hello()
    print('Hello, world!')
end

Call hello from C:

lua_getglobal(L, "hello");
lua_pcall(L, 0, 0, 0);
Ideas for Users

- Modifying software written in C is hard for users
- Give users the power to modify and extend the system
- Let users explore the system in an easy way
Ideas for Developers

- "Rapid Application Development" approach to driver/kernel development
- Modifying the system behaviour
- Configuration of kernel subsystems
Alternatives

- Python
- Java
Python

- Not to difficult to integrate in C
- Huge library
- Memory consumption
- Difficult object mapping
Use Cases

Java

- Easy to integrate
- Difficult object mapping
- Memory considerations
- Has been used for driver development
Lua in NetBSD Userland

- Part of the NetBSD base install since NetBSD 6
- Library (liblua.so) and binaries (lua, luac)
- Bindings to GPIO and sqlite
Lua in the NetBSD Kernel

- Started as GSoC project, porting Lunatik from Linux to NetBSD
- Proof that the Lua VM can run in the kernel, lack of infrastructure
- Infrastructure has been added
- About to be added to NetBSD -current
- Should be part of NetBSD 7
Source Code Layout

- Lua source code will be move from `src/external/mit/lua` to `src/sys/external/mit/lua`
- Kernel parts will reside under `src/sys/lua`
- The last step will be an update to the latest Lua (5.2.1)
Running in Userland

- Every process has its own address space
- Lua states in different processes are isolated
Running in the Kernel

- One address space
- Every thread that „is in the kernel“ uses the same memory
- Locking is an issue
The Big Picture

Userland

modload(8)
luactl(8)
sysctl(8)
luac(1)

Kernel

Compiler

Runtime

Lua
Lua

Kernel subsystems (Users)

Lua bindings

lua(4)

ioctl(2)
sysctl(3)

Filesistem
The lua(4) Device Driver

Userland

Kernel

Compiler

Runtime

Lua

Lua

lua(4)

ioctl(2)

sysctl(3)

Filesystem

Lua in the NetBSD Kernel

Marc Balmer <mbalmer@NetBSD.org>
lua(4) ioctl(2) / sysctl(3) Interface
Lua States
Lua Modules

Userland

Kernel

Filesystem

modload(8)

Lua bindings
Lua Users

- Userland
- Kernel
  - Kernel subsystems (Users)
- Filesystem
The luactl(8) Command
„require”

- Modules are kernel modules
- ’require’ can be turned off
- Modules must be assigned to Lua states by hand then
- By default ’require’ is on and modules are even autoloaded
- Module autoloading can be turned off
sysctl(8) Variables

- kern.lua.require=1
- kern.lua.autoload=1
- kern.lua.bytecode=0
- kern.lua.maxcount=0
- kern.lua.protect=0
Loading Lua Code
Security

- New Lua states are created empty
- Full control over the loading of code
- No access to kernel memory, -functions but through predefined bindings
- Dangerous code can be disabled at the byte-code level (prevention of endless loop DoS etc.)
Time for Questions