



ZOOM OUT BREW 2008 CONFERENCE

Introduction to Lua

Jack Pham, Senior Engineer
QUALCOMM Incorporated

QUALCOMM



Agenda

- What is Lua?
- BREW[®] Mobile Platform, IDL and Lua
- Extending Lua
- Q&A

What Is Lua?

- Lightweight, embeddable, extensible scripting language
- www.lua.org
- Tecgraf and Lablua at PUC-Rio, Brazil
- Roberto Ierusalimschy, *Programming in Lua*, 2nd Edition
- “moon” in Portuguese





Lua Features

- Lightweight: ~170KB on ARM
- Dynamically typed
 - Types: nil, boolean, number, string, userdata, function, thread, table
- Garbage collected
- Single data structure: table
- Metatables
 - Can be used to implement OOP
- First-class functions



Why Lua?

- Or any dynamic language...
- Prototyping, RAD
- Easier to learn
- Data-descriptive language / semantic-descriptive data
- Slow? Yes but...
 - More productive
 - More expressive
 - More flexible

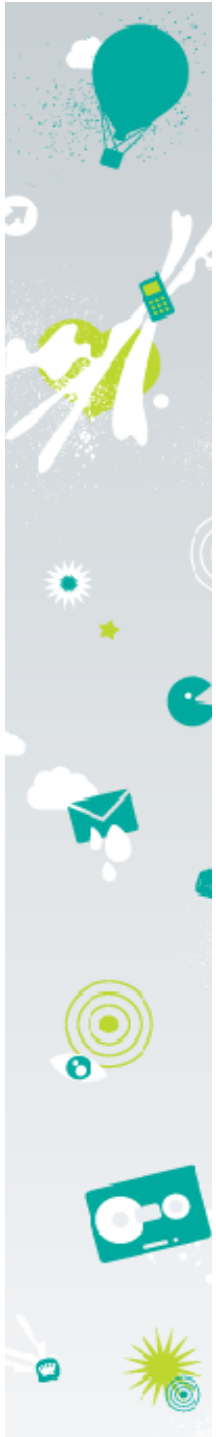


Lua in Use

- In Games
 - Scripting for game logic, state, customization
- In Applications
 - Adobe, Cisco, lighttpd, nmap
- In Qualcomm
 - ILua: BREW interface
 - uiOne™ solution: TrigML™ language meets Lua
 - PROG-401: Advanced TrigML Development with Lua
 - PROG-503: Debugging TrigML and Lua
 - BREW UI Widgets Theme Compiler
 - BREW CIFC compiler

Hello, World!

```
print("Hello, world!")
```





Another Example

```
-- define a factorial function
function fact(n)
  if (n == 0) then
    return 1
  else
    return n * fact(n-1) -- not a tail call
  end
end
```

```
fact(8) -- returns 40320
```



Chunks

- Sequence of Lua statements
- Loaded as a function
- “main” chunk
- e.g. dofile()

```
function dofile(filename)
    local f = assert(loadfile(filename))
    return f()
end
```

Functions and Closures

- First-class objects in Lua
 - Functions as arguments, return values

- E.g. *map* function

```
function square(n) return n*n end
```

```
function map(f, t)
```

```
    local t2 = {}
```

```
    for k, v in pairs(t) do t2[k] = f(v) end
```

```
    return t2
```

```
end
```

```
a = map(square, {1, 2, 3, 4, ...}) -- {1, 4, 9, 16, ...}
```

Functions and Closures

- Anonymous functions
- Lexically scoped

```
a = 10
function f()
  local a = 20
  return function() -- bind with 'a', this is a closure
    print(a)        -- 'a' is an upvalue
  end
end

g = f()
g() -- prints 20
```



The Table

- One and only Lua data structure type
- Can implement arrays, lookup tables, sets, records, lists, stacks, queues, etc.
- Contains array and dictionary portions
- Environments are tables too!
 - Variables are keys
 - `_G` is global environment table
- Can be used for namespaces

Table Examples

- Table as an array

```
a = { "one", "two", "three" } -- 1-based
```

```
print(#a) -- prints 3
```

```
a[4] = "four" -- numeric index
```

```
print(#a) -- prints 4
```

- Equivalent to

```
a = { [1]="one", [2]="two", [3]="three" }
```

Table Examples

- Table as a dictionary

```
d = { one=1, two=2, ["three"]=3 }
```

```
d["four"] = 4 -- string as key
```

```
d. five = 5 -- record-style key
```

```
d. one = nil -- remove entry
```

Table Examples

- Table as a namespace

```
Poi nt = { }
```

```
Poi nt. new = functi on(x, y) return {x=x, y=y} end
```

```
Poi nt. add = functi on(a, b) return {x= a. x + b. x,  
                                     y= a. y + b. y}
```

```
end
```

```
A = Poi nt. new(5, 5)
```

```
B = Poi nt. new(10, 20)
```

```
C = Poi nt. add(A, B) -- C: (15, 25)
```



Metatables

- Alter behavior for tables and userdata
- Useful for user-defined types and operations
- Metamethods:
 - Arithmetic (+, -, *, /)
 - Relational (<, <=, >, >=, ==, ~=)
 - Indexed access/assignment (table[key])
 - Garbage collection (only userdata)



Lua and BREW Mobile Platform

- ILua – BREW interface
 - Presents API similar to that of Lua C API functions
- IDL introduced in BREW
 - Define *remotable* interfaces for use across process boundaries
 - Primary mechanism for ILua->BREW invocation
 - c.f. IRemoteObject::Invoke()
 - idl2lua compiler generates .lua to include in your module
- No need to manage AddRef and Release!



IDL Example

IBankAccount.idl

```
interface IBankAccount : IQI
{
    /* AddRef, Release, QueryInterface
       inherited from IQI */
    AEResult Deposit(in Long amount);
    AEResult Withdraw(in Long req,
                     rout Long amt);
    readonly attribute Long Balance;
};
```



IDL/Lua Example

- IBankAccount.idl → IBankAccount.lua
- `idl2lua.exe -I $(incpaths) IBankAccount.idl`



IDL/Lua Example

- Simplified version of IBankAccount.lua

```
return {  
  AEEIID_IBankAccount = 0x1234ABCD,  
  IBankAccount = {  
    Deposit = function(self, amount)  
      self:Invoke(1, {pack("l", amount)})  
    end,  
    Withdraw = function(self, req)  
      return self:Invoke(2, ...)  
    end,  
    GetBalance = function(self)  
      return self:Invoke(3, ...)  
    end  
  }  
}
```

IDL/Lua Example

```
cs = require "ComponentServices"
idl = cs.RegisterInterfacesFromFile("IBankAccount.lua")
account = cs.CreateInstance(clsid, idl.AEEIID_IBankAccount)

account:Deposit(500)
cash = account:Withdraw(50)
assert(cash == 50)
assert(account:GetBalance() == 450)

account:Deposit(200)
assert(account.Balance == 650) -- attribute access as member
account.Balance = 1000000      -- error, read-only attribute

account = nil                  -- garbage-collected & calls Release()
```



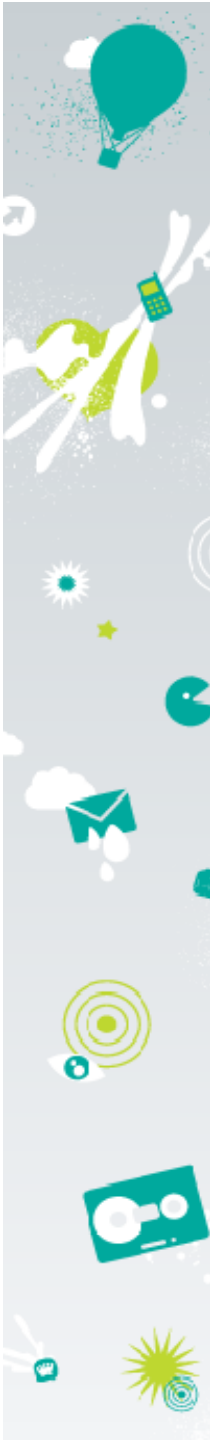
Extending Lua

- Lua C API: lua.h
- BREW: AEELua.h
 - e.g. lua_tonumber() → ILua_ToNumber()
- C as Library: C functions called from Lua
 - Lua as “glue” logic
- C as App: calls Lua code

Lua Stack

- Values accessed in C and Lua via virtual stack

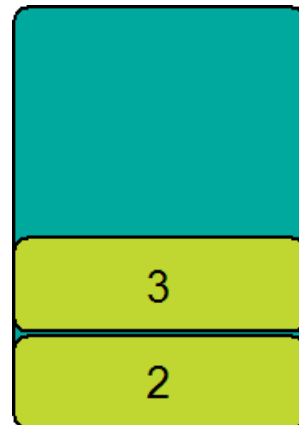
| Lua → C (inspection) | Lua → C (conversion) | C → Lua |
|---------------------------------|---------------------------------|-------------------------------------|
| ILua_IsBoolean | ILua_ToBoolean | ILua_PushBoolean |
| ILua_IsNumber ILua_IsInteger | ILua_ToNumber ILua_ToInteger | ILua_PushNumber ILua_PushInteger |
| ILua_IsString | ILua_ToString | ILua_PushString |
| ILua_IsUserData | ILua_ToUserData | ILua_NewUserData |
| ILua_IsFunction ILua_IsNil | | ILua_PushCFunction ILua_PushNil |



Lua Stack

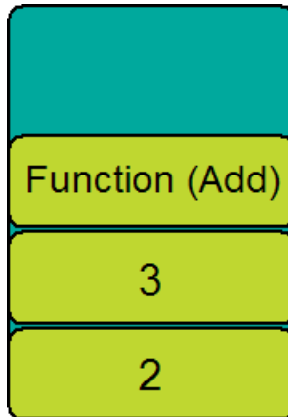


`lua_pushinteger(pi Lua, 2);`



`lua_pushinteger(pi Lua, 3);`

Lua Stack



```
lua_pushcfunction(lua, Add);
```

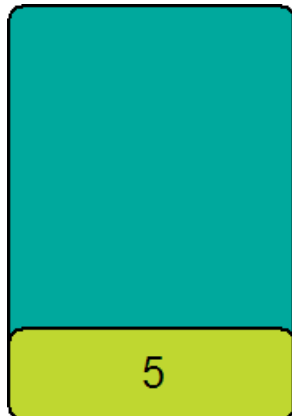
```
-- 2 arguments, 1 return
```

```
lua_call(lua, 2, 1);
```

```
-- -1 is top-of-stack
```

```
sum =
```

```
lua_tointeger(lua, -1);
```





User-Defined Types

- New types in C represented as Lua userdata
- `lua_newuserdata()` creates block of memory
- Write functions that operate on this type
- Use metatables (`__index` metamethod) to bind operations as OO-style methods



Userdata Example

```
struct Point { int x, y; }

// function Point.new(x, y)
int Point_New(ILua *pi Lua)
{
    Point *p = (Point*)ILua_NewUserData(pi Lua,
    sizeof(Point));
    // disclaimer: these should be type checked
    p->x = ILua_ToInteger(pi Lua, 1);
    p->y = ILua_ToInteger(pi Lua, 2);
    return 1;    // 1 stack value to return
}
```



Userdata Example

```
// function Point.add(p1, p2)
int Point_Add(ILua *pi Lua)
{
    // disclaimer: these should be type checked
    Point *p1 = (Point*)ILua_ToUserData(pi Lua, 1);
    Point *p2 = (Point*)ILua_ToUserData(pi Lua, 2);
    Point *p = (Point*)ILua_NewUserData(pi Lua,
    sizeof(Point));
    p->x = p1->x + p2->x;
    p->y = p1->y + p2->y;
    return 1;    // 1 stack value to return
}
```

Userdata Example

```
const AEE LuaRegFunc funcs[] = {  
    {"new", Point_New},  
    {"add", Point_Add},  
    {0, 0}  
};  
  
lua_registerLib(L, "Point", funcs);
```

Userdata Example

- Lua:

```
p1 = Point.new(4, 5)
```

```
p2 = Point.new(6, 3)
```

```
p3 = Point.add(p1, p2)
```

```
-- haven't defined __toString yet
```

```
print(p3)           -- userdata: 0x80126534
```



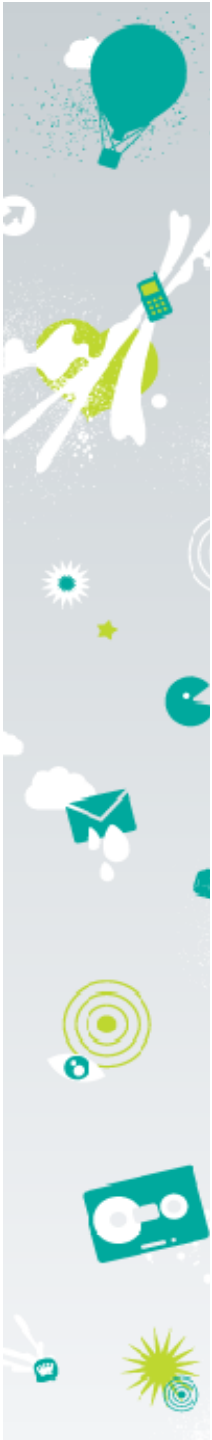
Userdata Example

- Should also create a metatable
- Set `__index` to bind objects with methods
 - Supports object-oriented style:
`p3: add(p1) -- sugar for p3.add(p3, p1)`
- Set `__tostring` for pretty printing
 - Before:
`print(p3) -- userdata: 0x80126534`
 - After:
`print(p3) -- Point: (10, 8)`



Start Using Lua!

- Write app code in Lua
 - Including TrigML apps
 - PROG-401: Advanced TrigML Development with Lua
 - PROG-503: Debugging TrigML and Lua
- Write components in Lua & IDL
 - Can be called from either Lua or C/C++
- Write libraries in C for Lua
 - Wrappers around legacy and non-IDL code
 - e.g. porting third-party code



Q&A