bbv
MAKING VISIONS WORK.

BOOKLET

EMBEDDED SCRIPTING MIT LUA

PROFITIEREN SIE VON UNSERER ERFAHRUNG!

Kontakt Schweiz
bbv Software Services AG
Blumenrain 10
6002 Luzern
Telefon: +41 41 429 01 11
E-Mail: info@bbv.ch

Kontakt Deutschland
bbv Software Services GmbH
Agnes-Pockels-Bogen 1
80992 München
Telefon: +49 89 452 438 30
E-Mail: info@bbv.eu

INHALT

1 EINLEITUNG

Software muss immer flexibler werden, dies gilt auch für den Bereich Embedded. Zunehmend läuft Software auf 32-bit-Systemen und hat immer mehr Anforderungen zu erfüllen. Es besteht nicht nur der Wunsch nach agiler Softwareentwicklung, die Software selbst soll auch auf einfache Weise erweiterbar sein, sei es zu Testzwecken oder um die Funktionalität mit geringem Aufwand an den Anwendungsfall anzupassen.

An dieser Stelle zeigen eingebettete Scriptsprachen ihre Stärken. Sie können nahtlos mit dem Rest der Software kommunizieren, die in Embedded Systemen traditionell entweder in C oder C++ entwickelt ist.

Dieses Booklet gibt einen Einblick in die Möglichkeiten, die eine eingebettete Scriptsprache, in diesem Fall Lua (1), bietet. Es kann jedoch nicht auf alle Möglichkeiten eingegangen werden, dies würde den Rahmen bei Weitem sprengen. Vielmehr ist dieses Booklet als praktische Einführung gedacht. Die hier enthaltenen praktischen Beispiele setzen Grundkenntnisse auf Einsteigerniveau von C++ voraus.

2 EINBETTEN VON SCRIPTSPRACHEN

Eingebettete Scriptsprachen gibt es viele, auch ihr Einsatzzweck ist vielfältig. Betrachtet man ihre Verbreitung, werden sich wenige kommerzielle Applikationen finden, die keine Scripting-Schnittstelle bieten.

Es sind Softwares aus allen Bereichen, die offensichtlich oder auch versteckt Scripting verwenden (hier keine abschliessende Liste, mehr unter (2)):

- Kommerzielle und OpenSource-Applikationen:
  Adobe Photoshop Lightroom, MySQL Workbench, VLC Media Player, Wireshark, etc.

- Spiele:
  World of Warcraft, Angry Birds, Civilization V, Crysis, Fable 2, etc.

- Andere:
  Lego Mindstorms NXT, MediaWiki (Templates), nginx (Web-server), Rockbox (Mediaplayer), etc.

In diesem Booklet wird Lua verwendet. Die folgenden Kapitel geben eine Einführung in die Sprache und beschreiben deren Einbettung.

3 LUA

Lua (portugiesisch: Mond) wurde an der PUC-Rio (Pontifical Catholic University of Rio de Janeiro, Brasilien, siehe (3)) im Jahre 1993 begonnen und seither weiterentwickelt. Ihr Einsatz ist weit verbreitet, und es existiert eine sehr grosse Anwendergemeinschaft.

Geschrieben ist Lua in C. Sie ist portabel, effizient, einfach zu verwenden und hat einen sehr kleinen Footprint. Der Footprint wird im Wesentlichen durch die Konfiguration und Ressourcen bestimmt, die zur Laufzeit verwendet werden. Zudem war eines der grundlegenden Ziele die einfache Integration in C und C++.

3.1 EINSATZZWECK

Lua kann in sehr unterschiedlichen Situationen eingesetzt werden. Dazu gehören einfache Dinge wie lesen von Konfigurationsfiles oder als flexibles Testinterface. Darüber hinaus kann Lua auch viel intelligentere Aufgaben übernehmen wie beispielsweise Server-Side-Scripting bei Embedded Webservern oder Steuerung des gesamten User-Interfaces von Geräten wie LCD und Knöpfe. An allen Stellen, die nicht kritisch auf Performance (vor allem CPU) bezogen sind, kann Lua eingesetzt werden, vorausgesetzt, es existiert ein geeignetes Interface. Vor allem High-Level-Funktionen können effizient von Lua aus verwendet werden.

3.2 LIZENZ

Lua wird unter der MIT License (siehe (4)) vertrieben. Diese ist sehr offen, und es ist deshalb auch einfach möglich, Lua den Bedürfnissen des jeweiligen Projekts anzupassen.

Die wesentlichen Punkte der MIT License sind:

- Ungehindertes Kopieren, Abändern, Vertreiben.
- Keine Einschränkungen für kommerzielle Software.
- Keine Lizenzkosten.
- Kein Zwang, eigene Veränderungen am Sourcecode zu veröffentlichen.
- Es wird keine Haftung von PUC-Rio übernommen.

Achtung:

Diese Liste ist nicht rechtsgültig, siehe Originallizenz (4).

3.3 SPRACHE

Die Sprache ist nach dem Motto easy to learn, hard to master sehr einsteigerfreundlich, bietet aber eine enorme Flexibilität.

Die unkomplizierte Syntax verhindert weitgehend obfuscated code, der kaum mehr lesbar ist. Der Vorfahre im Geiste ist Lisp. Lua hat jedoch eine Syntax, die mehr an C und Pascal erinnert.

Die grundlegende und einzige Datenstruktur, die Lua kennt, ist die assoziative Tabelle. Diese kann neben gewohnten keyvalue Paaren auch rein numerische Werte als Indices verwalten.

Jede Tabelle hat eine metatable, welche die Verwaltung der Tabelle übernimmt. Somit kann die Verwendung einer Tabelle bei Bedarf den Bedürfnissen angepasst werden, z. B. kann eine entsprechende metatable Schreiboperationen auf eine Tabelle verhindern.

In den folgenden Unterkapiteln werden die wichtigsten Sprachelemente aufgeführt, nicht jedoch die Sprache als Ganzes dargestellt. Dazu sei auf (5) verwiesen.

3.3.1 Kommentare

In Lua werden Kommentare mit zwei Strichen gekennzeichnet, die einen Kommentar bis zum Ende der jeweiligen Zeile bezeichnen. Lange Kommentare können mehrzeilig sein und sind mit entsprechenden Klammern gekennzeichnet. Beispiele:


```
-- this is a comment
print('Hello World!') -- prints a text to console
--[[ start of long comment
multiline comment
```

3.3.2 Typen

Lua ist dynamisch schwach typisiert, folgende Typen stehen zur Verfügung:

- **`nil:`** Nullelement
- **`boolean:`** Boolsche Werte true und false
- **`number:`** Numerische Werte, rationale Zahlen
- **`string:`** Zeichenkette
- **`able:`** Tabelle, Metatabelle
- **`function:`** Funktionen (Lua wie auch C)
- **`userdata:`** Benutzerdaten, die Lua nur als Zeiger kennt, werden jedoch vom garbage collector aufgeräumt
- **`ightuserdata`**: analog zu userdata, aber ohne Metatable und werden auch vom garbage collector nicht aufgeräumt
- **`thread:`** Thread

Zur Laufzeit liefert die Funktion string = type(var) eine textuelle Repräsentation der angegebenen Variable bzw. des Objekts.

3.3.3 Tabellen

Es ist die einzige Datenstruktur, die Lua kennt. Sie ist assoziativ und kann Hierarchien bilden. Sie können als Funktionsargumente wie auch als Rückgabewerte dienen. Eine Tabelle zu definieren ist sehr einfach:

```
tab =
{
    a = 1, b = 2, c = „string“,
    [1] = „hello“,
    [2] = „world“,
    nested = { a = 10, b = 20 },
}
```

Als Werte kommen alle Lua bekannten Typen in Frage. Um auf die einzelnen Einträge zuzugreifen, gibt es folgende Möglichkeiten, die für Lua gleichbedeutend sind:

```
var_a = tab.a
var_b = tab['b']
var_str = tab[1] .. „ „ .. tab[2]
tmp_a = tab.nested.a
tmp_b = tab['nested'].b
```

Sofern nichts anderes definiert ist, kann eine Tabelle dynamisch wachsen. Einträge werden ungültig, indem sie auf nil gesetzt werden.

Die Verwendung im Zusammenhang mit Funktionen ist intuitiv (auch wenn an dieser Stelle bezüglich Funktionen vorgegriffen wird):


```
function swap(tab)
    return { a = tab.b, b = tab.a }
end
result = swap({ a = 10, b = 20 })
```

3.3.4 Kontrollstrukturen

Kontrollieren den Programmfluss, dazu gehören Bedingungen, Schleifen und Funktionen.

3.3.4.1 Funktionen

Diese werden im Allgemeinen so definiert:

```
function name(arg1, arg2)
    block
end
```

Die Anzahl Parameter variiert zwischen 0 und vielen. Eine variable Anzahl von Argumenten ist ebenfalls unterstützt:

```
function name(...)
    local arg = {...}
    -- variable arg is a table containing arguments
    -- having an attribute n, stating number of arguments
end
```


Bei variabler Anzahl Parameter existiert eine lokale Variable arg als Tabelle, welche alle Argumente enthält. Beispiel:

```
function find(i, ...)
    return arg[i]
end
result = find(2, 10, 20, 30, 40) -- result == 20
```

Die obigen Beispiele zeigen die geläufige Art, Funktionen zu definieren. Da Funktionen im Grunde ganz normale Datentypen sind, können sie auch wie solche zugewiesen werden:

```
name = function(arg1, arg2)
    block
end
```


3.3.4.2 Bedingte Ausführung

Lua kennt nur eine Kontrollstruktur, die bedingte Ausführung erlaubt. Diese wird mit **`if ... then ... end`** umgesetzt. Dabei sind mehrfache Bedingungen möglich:

```
if condition1 and condition2 then
    block
elseif condition3 or condition4 then
    block
elseif not condition5 then
    block
else
    block
end
```

Einen Mechanismus wie switch (in C und C++) gibt es in Lua nicht. Zu beachten gilt es auch: Die einzelnen Bedingungen brauchen nicht zwingend runde Klammern, jedoch wird empfohlen, sie zu schreiben. Gerade bei komplexen Bedingungen dienen Klammern dem Verständnis.

Neben **`and`**, **`or`** und **`not`** existieren folgende Vergleichsoperatoren:
< / > / <= / >= / ~= / ==

3.3.4.3 Schleifen

Wie in nahezu jeder anderen Sprache gibt es verschiedene Schleifen, um einen Block wiederholt auszuführen.

while hat die Abbruchbedingung am Anfang, die Schleife wird unter Umständen nicht durchlaufen:

```
while expression do
    block
end
```

repeat prüft das Abbruchkriterium am Schluss, der Block wird mindestens ein Mal durchlaufen:

```
repeat
    block
until expression
```

for ist die flexibelste Schleife, welche nicht nur numerische Grenzen, sondern auch Iteratoren über Mengen haben kann:

```
for var = startvalue,endvalue,step do
    block
end
```

wobei **step** auch weggelassen werden kann. Die andere Verwendung von **for** ist:

```
for var_1,..,var_n in iterator do
    block
end
```


Dazu ein Beispiel:

```
function index_of(tab, text)
    for i,s in ipairs(tab) do
        if s == text then
            return i
        end
    end
    return -1
end
local index = index_of({„dog", „cat", „mouse"}, „cat")
```

Gemeinsamkeiten aller Schleifen (gleiches Verhalten wie in C und C++):

- **`break`**: beendet die Schleife sofort und führt das Programm nach dem Scope der aktuellen Schleife fort.
- **`continue:`** beginnt erneut am Anfang der Schleife, ohne die Schleifenvariable(n) neu zu initialisieren. Der Rest des verbleibenden Blocks wird übersprungen.

3.3.5 Metatabellen

Immer wenn Lua auf eine Tabelle zugreift, verwendet es deren metatable. Darin sind Funktionen definiert, die bestimmen, in welcher Art auf die Tabelle zugegriffen werden kann.

Die Metatable besteht aus folgenden Funktionen:

- **`__add`**: Operation a + b (Addition)
- **`__sub`**: Operation a – b (Subtraktion)
- **`__mul`**: Operation a * b (Multiplikation)
- **`__div`**: Operation a/b (Division)
- **`__mod`**: Operation a%b (Modulo-Division)

- **__pow**: Operation x^y (Power)
- **__unm**: Operation –x (Unäre Minusoperation)
- **__concat**: Verbindungsoperation ..
- **__len**: Längenoperation #
- **__eq**: Operation == (Vergleich auf Gleichheit)
- **__lt**: Operation < (Vergleich auf Kleiner)
- **__le**: Operation <= (Vergleich auf Kleiner oder Gleich)
- **__index**: Zugriffsoperation table[key] für bestehende key Werte
- **__newindex**: Zugriffsoperation table[key] für neue key Werte
- **__call**: Aufruf einer Funktion

Es müssen nicht immer alle Funktionen definiert sein. Folgendes Beispiel soll die Verwendung von Metatabellen verdeutlichen:

```
mt = {
     __add = function(val1 , val2)
         return { a = val1.a + val2.a }
     end
}
tab1 = { a = 10 }
setmetatable(tab1 , mt)
tab2 = { a = 20 }
setmetatable(tab2 , mt)
tab3 = tab1 + tab2 -- call to __add , tab3.a is now 30
print(tab3.a)   -- prints 30
```

Es ist durchaus möglich, einen einzigen Eintrag in der Metatable zu ersetzen, falls derjenige schon existiert. Dies ist ein Beispiel, wie über die Metatable auf die eigentliche Tabelle zugegriffen wird:


```
tab = { a = 99 }
-- 'tab' gets metatable , somehow
getmetatable(tab).__index = function(table , key)
    if (key == 'a') then
        return table.a
    else
        error('access invalid key')
    end
end
```

Mit einer geeigneten Implementation von newindex wird der Zugriff auf die Tabelle für neue Tabelleneinträge kontrolliert. Beim Versuch, neue Einträge in die Tabelle zu schreiben, bricht der Fehler error im else-Zweig ab.

```
tab = { a = 5 }
setmetatable(tab, {
    __newindex = function(table , key, value)
        error('no new table entries allowed')
    end
})
tab.a = 10 -- a existed before
tab.b = 20 -- fails because of error in __newindex
```


3.3.6 Closures

Als Closure wird eine Programmfunktion bezeichnet, die ihren Entstehungskontext kennt. Das folgende Beispiel soll dies verdeutlichen:

```
function create_counter()
    local value = 0
    return function()
        value = value + 1
        return value
    end
end

c1 = create_counter()
c2 = create_counter()
print(c1()) -- prints 1
print(c1()) -- prints 2
print(c2()) -- prints 1
print(c2()) -- prints 2
```

Die Funktion **`create _ counter`** liefert eine Funktion, die jeweils den nächsten Zählerwert zurückliefert, dabei erhöht sie jeweils den Wert der Variablen **`value`**. So ist es auf einfache Art möglich, unabhängige Zähler (**`c1`** und **`c2`**) zu haben, die jeweils einen eigenen Zählerstand verwalten.

Die Closure zeigt sich nun darin, dass die Variable value nicht innerhalb des Funktionsblocks liegen darf, die den Wert zurückliefert, sonst würde sie bei jedem Aufruf neu initialisiert:

```
function create_counter()
    return function()
        -- initialized by every call
        -- to the counter , always 0
        local value = 0
        value = value + 1
        return value
    end
end
```

Bei jedem Aufruf käme der gleiche Wert (1) als Resultat der Funktion heraus.

Im globalen Raum darf die Variable value auch nicht liegen, sonst würden sich die unterschiedlichen Zähler gegenseitig beeinflussen:

```
value = 0
function create_counter()
    return function()
        -- all counter access global value
        value = value + 1
        return value
    end
end

c1 = create_counter()
c2 = create_counter()
print(c1()) -- value == 1
print(c2()) -- value == 2
print(c1()) -- value == 3
print(c2()) -- value == 4
```

Dies entspricht nicht dem erwarteten Verhalten der (unabhängigen) Zähler.

3.4 LIBRARIES

Lua liefert als Grundpaket eine ganze Reihe von Bibliotheken (libraries), um die häufigsten Bereiche abzudecken.

- **`bit32`**: Bitoperation, ausgerichtet auf 32-bit-Plattformen
- **`coroutine`**: Verwaltung von coroutinen
- **`debug`**: Zusatzfunktionen zum Tracen und Debuggen
- **`file`**: Dateioperationen
- **`io`**: Input/Output-Operationen
- **`math`**: Grundlegende Mathematik-Operationen, mit etwa dem gleichen Umfang der C-Standard-Mathematik-Library
- **`os`**: Betriebssystemoperationen, plattformagnostisch
- **`package`**: Verwaltung von zuladbaren Modulen (packages), sofern dies von der Plattform unterstützt wird
- **`string`**: Stringverarbeitung wie Formatieren, Suchen und Ersetzen
- **`table`**: Allgemeine Tabellenverwaltungsfunktionen

Oft wird nur ein Teil der Libraries verwendet, um die Möglichkeiten der Lua-Programme besser kontrollieren (einschränken) zu können. Ausblick auf das nächste Kapitel: Nach dem Erstellen eines Lua States müssen die Libraries dem State bekannt gemacht werden. Dabei gibt es zwei verschiedene Strategien:

1. Alle Libraries auf einen Schlag mit **`luaL_openlibs()`**
2. Libraries einzeln mit **`luaopen_*()`**, jeweils für die oben aufgeführten Libraries, z. B. **`luaopen_math()`**, **`luao-pen_string()`** etc.

Auf den Inhalt der einzelnen Libraries wird an dieser Stelle verzichtet, siehe dazu (5).

Garbage Collector

Lua besitzt einen automatischen Garbage Collector, welcher auch manuell angestossen werden kann. Alle Objekte, die von Lua nicht mehr erreicht werden können, löscht der Garbage Collector bei seinem nächsten Durchgang. Dies betrifft alle speicherbasierten Typen, wie **`string`**, **`table`**, **`userdata`**, **`function`**, **`thread`**, sowie interne Strukturen.

Implementiert ist der Garbage Collector als mark-and-sweep-Algorithmus.

Die Häufigkeit und Agressivität des Collectors kann mit der Funktion **`lua_gc(...)`** konfiguriert werden.

Folgend ein Beispiel:

```
local tab = { a = 10 }
-- ...
tab = 20 -- later
```

Die Variable tab repräsentiert zunächst eine Tabelle. Später wird diese mit einem anderen Wert, in diesem Fall einem Skalar, überschrieben. Die Tabelle wird nirgends referenziert und wird freigegeben. Der Garbage Collector löscht sie zu gegebener Zeit. Soll der Collector zu einer bestimmten Zeit angestossen werden, so ist die Funktion **`collcectgarbage()`** in Lua verfügbar, die dies erledigt. Aus C++ kann der Collector auch angestossen werden: **`lua_gc(LUA_GCCOLLECT)`**.

3.5 C API

In diesem Kapitel wird die Schnittstelle zwischen C/C++ und Lua erklärt. Immer wenn von C++ die Rede ist, funktioniert der exakt gleiche Mechanismus auch für C. Es wird hier allerdings darauf verzichtet, immer beide zu erwähnen.

Die Schnittstelle zwischen Lua und C++ ist simpel und effizient. Der grundlegende Mechanismus für den Austausch beider Welten ist ein Stack. Aus der Sicht von C++ wird mit Stack-Standardoperationen push und pop gearbeitet, aus Sicht von Lua ist dies automatisch gegeben.

Ein Lua-Programm läuft in einem sog. Lua State, welcher einen bestimmten Zustand innehat. Dieser besteht aus Daten wie auch dem für Lua nötigen inneren Zustand (Informationen über den Garbage Collector etc.). Der Lua State wird entweder vom Lua-Programm oder über das C API verändert.

Es können problemlos mehrere Lua States parallel existieren, diese wissen allerdings nichts voneinander. Es gibt auch keinen automatischen Datenaustausch zwischen den einzelnen Lua States. Jeder State ist quasi eine Sandbox.

Aus Sicht von C++ gibt es das normale API (Funktionen mit Präfix lua_) sowie die auxiliary library (Funktionen mit Präfix `luaL_`), welche die Verwendung des API einfacher gestalten.

Dieses Kapitel behandelt nicht das gesamte API, sondern zeigt anhand kleiner Beispiele dessen Funktionsweise.

An dieser Stelle wird darauf verzichtet, die einzelnen Schritte zum Übersetzen von Programmen aufzuführen.

Die Beispiele sind einfach genug, um mit einem beliebigen C++-Compiler (GCC, Visual Studio etc.) klarzukommen.

3.5.1 Lua States

Ein Lua State wird representiert durch eine Datenstruktur, das API stellt alle nötigen Funktionen zur Verfügung, um diesen zu verwalten.

```
lua_State * lua = luaL_newstate();
// ...
lua_close(lua);
```

Solange nun der State gültig ist, kann mit ihm kommuniziert werden. Zu beachten gilt: Ein neu erstellter Lua State ist immer leer. Es werden nicht automatisch Code ausgeführt oder Variablen gesetzt.

3.5.2 Ausführen von Lua-Programmen

Das wohl einfachste C++-Programm, das Lua einbettet, ist:

```
#include <lua.hpp>
#include <cstdlib>

int main(int, char **)
{
    lua_State * lua = luaL_newstate();
    luaL_openlibs(lua);
    luaL_dostring(„print('Hello World!')“);
    lua_close(lua);
    return EXIT_SUCCESS;
```


Sofern nichts anderes konfiguriert (die Defaultkonfiguration enthält sowohl Interpreter als auch den Compiler) ist, schreibt dieses einfache Programm Hello World! auf die Konsole.

Analog könnte man auch **luaL_dofile**(...) verwenden, um ein externes Lua-Programm auszuführen. **luaL_dofile** erkennt auch automatisch, ob die Daten aus der angegebenen Datei Lua-Sourcecode ist oder ob sie schon kompiliert wurden.

```
#include <lua.hpp>
#include <cstdlib>

int main(int, char **)
{
    lua_State * lua = luaL_newstate();
    luaL_openlibs(lua);
    luaL_dofile(„simple.lua“);
    lua_close(lua);
    return EXIT_SUCCESS;
```

Das zugehörige Lua-Programm:

```
print('Hello World!')
```


3.5.3 Aufruf von Funktionen

Der neu erzeugte Lua State oder auch ein zusätzliches Lua-Programm nützt in der Praxis nicht viel. Der Grund für das Einbetten von Lua in C++ ist im Normalfall das Bedürfnis, beide Welten zu vereinen und sie miteinander zu verwenden, um ein bestimmtes Ziel zu erreichen. Hier wird gezeigt, wie Funktionen, geschrieben in Lua und C++, gegenseitig verwendet werden. Dies anhand des einfachen Beispiels einer Addition.

Aufruf C++ nach Lua

Hier ein Lua-Programm, welches eine einfache Addition durchführt:

```
function add(a, b)
    return a + b
end
```

Diese Funktion ist innerhalb des Lua States global definiert. Solche global definierten Symbole können mit **`lua_getglobal`**(...) referenziert werden, d. h., wenn Lua das angegebene Symbol kennt, wird dessen Wert (für Tabellen und Funktionen die Referenz) auf den Stack gelegt.

Die Funktion add verlangt zudem zwei Operanden. Wie oben erwähnt, funktioniert die Kommunikation über einen Stack, d. h., die beiden Operanden wie auch das Resultat müssen auf diesen Stack geschrieben bzw. davon gelesen werden (hier aufgeführt sind lediglich die notwendigen Zeilen, das Erstellen des Lua States ist oben beschrieben).


```
double call_lua_add(lua_State * lua, double a, double b)
{
    // read global symbol onto the stack
    lua_getglobal(lua, „add“);

    // push operands to the stack
    lua_pushnumber(lua, a);
    lua_pushnumber(lua, b);

    // call function (2 arguments , 1 result)
    lua_pcall(lua, 2, 1, 0);

    // read result from stack
    double result = lua_tonumber(lua, -1);

    // remove result from stack
    lua_pop(lua, 1);
    return result;
}
```

Hier wird auch gleich eine der wichtigsten Eigenheiten dieser Kommunikation ersichtlich: Es obliegt der eigenen Verantwortung, richtig mit dem Stack umzugehen. Ein stack overflow kann durchaus passieren, sofern der Stack nicht aufgeräumt wird. Dies ist nicht zu verwechseln mit dem in Lua vorhandenen Garbage Collector, da dieser nicht wissen kann, welche Absichten die C++-Welt mit dem Stack verfolgt.

Eine weitere Erkenntnis, die bereits erwähnt wurde, ist in diesem Beispiel ersichtlich: Stackoperationen werden nur in C++ verwendet, der Stack ist für die Lua-Welt transparent.

Aufruf Lua nach C++

Nun der umgekehrte Fall. Ein Lua-Programm möchte eine Funktion verwenden, die in C++ geschrieben ist, wieder eine Addition. Es soll folgendes Lua-Programm

```
function sum(a, b, c)
      return add(a, add(b, c))
end
```

die Funktion add (implementiert in C++) verwenden, um das gewünschte Resultat zu berechnen. Dazu benötigen wir eine entsprechende Funktion in C++, die genau wie oben den Stack für den Austausch der Operanden und Resultate mit dem Lua State verwendet.

Die Signatur einer solchen Funktion muss der Konvention des C APIs von Lua folgen:

```
int add(lua_State * lua)
{
    // read operands from stack
    double a = lua_tonumber(lua, -2);
    double b = lua_tonumber(lua, -1);

    // push result to the stack
    lua_pushnumber(lua, a + b);

    // tell Lua how many result values there are
    return 1;
```

Zu beachten ist hier die umgekehrte Reihenfolge der Parameter **a** und **b**, wie sie auf dem Stack liegen. Da **b** nach **a** auf den Stack geschrieben wurde, liegt a auch auf einem Platz tiefer.

Zusätzlich: Der jeweilige Index bezieht sich auf die Position auf dem Stack. Ein positiver Index beschreibt eine absolute Position innerhalb des Stacks, ein negativer Index ist relativ zum stack top. –1 ist somit die zuoberst auf dem Stack liegende Zahl, gleich unterhalb des nächsten freien Platzes.

Zum Schluss das Exportieren der Funktion in den Lua State:

```
lua_register(lua, „add“, add);
```

Funktionen müssen nicht zwingend den gleichen Namen in C++ und Lua besitzen, es ist jedoch ratsam, um Verwechslungen zu vermeiden.

3.5.4 Globale Symbole

Skalare

Analog zur Verwendung von Funktionen kann auf die gleiche Weise eine globale Variable definiert, geschrieben und gelesen werden:

```
void set_lua_global(lua_State * lua,
        const char * name, double value)
{
    / push value to the stack
    lua_pushnumber(lua, value);

    // set global variable with Lua State ,
    // its value is expected to be on top of stack
    lua_setglobal(lua, name);
}
```

Existiert das angegebene Symbol bereits, wird der Wert überschrieben. Unter Umständen werden Daten nicht mehr verwendet und der Garbage Collector wird sie löschen.

Analog dazu das Lesen:

```
double get_lua_global(lua_State * lua, const char * name)
{
    // read global value
    lua_getglobal(lua, name);

    // read number from stack
    double result = lua_tonumber(lua, -1);

    // free stack from this operation
    lua_pop(lua, 1);
    return result;
```

Tabellen

Tabellen sind gegenüber Funktionen und skalaren Variablen ein wenig komplizierter. Es ist möglich, mit keyvalue-Paaren oder auch mit numerischen Indices auf eine Tabelle zuzugreifen.

Erzeugt werden kann eine Tabelle wie folgt (im globalen Namensraum des Lua States):

```
void create_table(lua_State * lua, const char * name)
{
    // create new (empty , no preallocated space)
    lua_newtable(lua);

    // store table using provided name
    lua_setglobal(lua, name);
```


Nun schreiben wir einen Wert (hier eine Zahl) als Tabelleneintrag in den Lua State. Dieser Zugriff auf die Tabelle erfolgt mit einem key value-Paar:

```
void set_table_value(lua_State * lua,
        const char * table_name,
        const char * key, double value)
{
    // get table reference onto stack
    lua_getglobal(lua, table_name);

    // push key,value pair
    lua_pushstring(lua, key);
    lua_pushnumber(lua, value);

    // write value from the top of stack into
    // the table , the key is expected to be
    // right below the value on the stack
    lua_settable(lua, -3);
```


Es ist auch möglich, in eine Tabelle zu schreiben unter Verwendung eines Indexes innerhalb der Tabelle:

```
void set_table_value_at(lua_State * lua,
        const char * table_name,
        int index, double value)
{
    // get global variable onto stack
    lua_getglobal(lua, table_name);

    // push index and value onto stack
    lua_pushinteger(lua, index);
    lua_pushnumber(lua, value);

    // write value into the table (currently
    // at top -3), index here means index within
    // table
    lua_settable(lua, -3);
}
```

Daten aus einer Tabelle zu lesen, ist ähnlich einfach. Auch hier ist es möglich, Daten über einen key oder über einen Index zu lesen (zur Erinnerung: Alle Tabellen in Lua sind assoziativ).


```
double get_table_value(lua_State * lua,
        const char * table_name, const char * key)
{
    // read reference to table onto stack
    lua_getglobal(lua, table_name);

    // get value from table to the stack
    lua_pushstring(lua, key);
    lua_gettable(lua, -2);

    // get value from stack and clean it up
    double result = lua_tonumber(lua, -1);
    lua_pop(lua, 1);
    return result;
```


Unter Verwendung von Indices ist der Zugriff auf die Tabelle ähnlich:

```
double get_table_value_at(lua_State * lua,
        const char * table_name, int index)
{
    // read reference to table onto stack
    lua_getglobal(lua, table_name);

    // get value from table to the stack
    lua_pushinteger(lua, index);
    lua_gettable(lua, -2);

    // get value from stack and clean it up
    double result = lua_tonumber(lua, -1);
    lua_pop(lua, 1);
    return result;
```

Zu beachten gilt: Die Funktionen **`lua_gettable`** und **`lua_settable`** bewirken, dass die Funktionen index bzw. newindex der Metatabelle verwendet werden. Will man dies umgehen, muss auf die Funktionen **`lua_rawset`** resp. **`lua_rawget`** zurückgegriffen werden. Dies kann für Spezialfälle sehr nützlich sein.

4 SWIG

Wird das Interface genügend komplex, ist es schwierig, es manuell zu implementieren. Es existieren verschiedene Bindings, welche C++-Strukturen in Lua abbilden und das Interface benutzerfreundlich umsetzen. Zu erwähnen sind hier Lua++, LuaBind oder SLB, siehe (6).

Einen anderen Ansatz verfolgt SWIG, der SoftWare Interface Generator, siehe (7).

Dieser erwartet eine Inferfacespezifikation und generiert entsprechenden Code. Das Einfache daran ist, dass SWIG C++ Headerfiles interpretieren kann. Somit muss ein Interface nicht manuell (nochmals) implementiert werden. Dies funktioniert übrigens auch objektorientiert.

SWIG ist sehr flexibel, der Generator kann auch Bindings für andere Sprachen (z. B. Python, Common Lisp, Perl, PHP, etc.) erstellen. Hier liegt sehr viel Potenzial, z. B. für Interfaces zu Testzwecken.

Die aufgeführten Beispiele verwenden SWIG und GCC und zeigen die nötigen Schritte auf der Kommandozeile (in diesem Fall bash auf Linux). Sie sind aber einfach genug, um problemlos auch auf Windows durchgeführt zu werden.

4.1 SIMPLES INTERFACE

Hier nochmals das Additions-Beispiel, dieses Mal unter Verwendung von SWIG, um das Interface nicht manuell zu implementieren. Das C++ Headerfile (**add.hpp**):

```
#ifndef __ADD__HPP__
#define __ADD__HPP__

double add(double, double);

#endif
```

Die Implementation von add.cpp ist rein prozedural. Die Funktion main ist nicht im Headerfile enthalten, sie muss auch nicht nach Lua exportiert werden. Achtung: Die nach Lua exportierten Funktionen dürfen nicht **static** sein.


```
#include „add.hpp“
#include „lua.hpp“
#include <cstdlib>

extern „C“ { int luaopen_ops(lua_State *); }

double add(double a, double b)
{ return a + b; }

int main(int, char ** argv)
{
    lua_State * lua = luaL_newstate();

    // register all standard libraries
    luaL_openlibs(lua);

    // register from SWIG generated library
    luaopen_ops(lua);

    // execute Lua program
    luaL_dofile(lua, argv[1]);
    lua_close(lua);
    return EXIT_SUCCESS;
```

Die Interfacespezifikation (**ops.i**) von SWIG hat eine eigene Syntax. Sie ist so aufgebaut, dass einfache Interfaces auch einfach spezifiziert werden können. Allerdings bietet sie ungeahnte Möglichkeiten, dies sogar unabhängig von der verwendeten Scriptsprache.


```
%module ops
%{
#include „add.hpp“
%}
%include „add.hpp“
```

Die Interfacespezifikation wird nun von SWIG dazu verwendet, Code zu generieren, der normal kompiliert und gelinkt wird.

```
$ swig -c++ -lua ops.i
$ g++ -c ops_wrap.cxx
$ g++ -c add.cpp
$ g++ -o add add.o ops_wrap.o -llua
```

So kann jedes Lua-Programm nun auch die exportierten Funktionen verwenden. Man beachte den Namensraum **ops**, welcher in der Interfacespezifikation als module angegeben wurde:

```
print(ops.add(10, 20))
```

Der grosse Vorteil zeigt sich dann, wenn mehrere Interfaces/Module gleichzeitig zum Einsatz kommen.

4.2 OBJEKTORIENTIERTES INTERFACE

Bis jetzt waren alle Interaktionen zwischen C++ und Lua rein prozedural. Dies ist für einfache Interfaces bzw. Problemstellungen unter Umständen ausreichend.

Bei mittleren und grösseren Projekten sind solche Interfaces meist nicht trivial, oft objektorientiert. Diese können mit einem geeigneten Binding, sogar auch manuell, durchaus umgesetzt werden. In solchen Fällen ist SWIG allerdings eine grosse Hilfe.

Das folgende Beispiel (implementiert in C++) soll nun zeigen, wie ein Interface mit einem objektorientierten Design verwendet werden kann.

In diesem Beispiel will man verschiedene LCDs aus einem Lua-Programm steuern. Die Implementation in C++ interessiert an dieser Stelle nicht, lediglich die Klassendefinitionen werden dargestellt. Das Beispiel implementiert eine Klassenhierarchie mit einer Basisklasse LCD, welche als Interface zu den verschiedenen LCDs dient. Das Lua-Programm möchte sich aber nicht um diese Details kümmern müssen, daher eine Funktion `create_lcd` (eine Factory), die ein entsprechendes Objekt erzeugt.


```
class LCD
{
    public:
        virtual ~LCD();
        virtual void show(const std::string &) = 0;
        virtual void backlight(bool) = 0;
};

class SmallLCD : public LCD
{
    public:
        virtual ~SmallLCD();
        virtual void show(const std::string &);
        virtual void backlight(bool);
};

class LargeLCD : public LCD
{
    public:
        virtual ~LargeLCD();
        virtual void show(const std::string &);
        virtual void backlight(bool);
```

Man beachte: Das SWIG-Interface kennt C++-Klassen bzw. Klassenhierarchien und kann diese auch in Lua abbilden. Lediglich die Klasse **`std::string`** aus der STL ist speziell zu behandeln. Zu diesem Zweck liefert SWIG alle nötigen Definitionen (**`std_string.i`**).

```
%module lcd;
%{
#include „LCD.hpp“
extern LCD * create_lcd(int);
%}
%include <std_string.i>
%include „LCD.hpp“
extern LCD * create_lcd(int);
```

Das Lua-Programm erzeugt über die Factory ein Objekt und verwendet dessen Schnittstelle (definiert durch die Basisklasse).

```
local display = lcd.create_lcd(0)
display:show('Hello World!')
display:backlight(true)
```

Die Implementation von **`create_lcd`** ist möglichst einfach gehalten und hier nur vollständigkeitshalber aufgeführt:

```
LCD * create_lcd(int type)
{
    switch (type) {
        case 0: return new SmallLCD;
        case 1: return new LargeLCD;
    }
    return NULL;
```


5 EINSATZBEISPIELE

In diesem Kapitel werden einige Beispiele aufgezeigt, in welcher Art Lua zum Einsatz kommen kann. Dies ist natürlich keine abschliessende Liste.

5.1 KONFIGURATIONSFILES

Viele Systeme werden über entsprechende Files (z. B. XML, INI, etc.) konfiguriert. Diese müssen (mindestens) gelesen werden können. Dazu wird oft eine bestehende Library verwendet (z. B. **`libxml2`**).

Mit dem Compiler von Lua ist es auf einfache Weise möglich, ein in Lua geschriebenes Konfigurationsfile zu verarbeiten und die Daten aus dem Lua State zu lesen.

Falls das File korrupt ist, wird es der Compiler merken, es braucht keine zusätzliche Fehlererkennung.

Auch kann der Einsatz von Lua Vorteile bringen, da z. B. **`libxml2`** oder **`xerces-c`** einen um vielfach grösseren Footprint haben.
Ein vorangegangenes Beispiel hat gezeigt, wie einfach es ist, Variablen aus einem Lua State auszulesen.

Einen zusätzlichen Bonus erhält man, wenn Lua-Programme zur Konfiguration ausgeführt werden sollen. Dann braucht es nur ein entsprechendes Interface.

Nun ein Beispiel einer (passiven) Konfiguration mit einem Lua-Sourcefile als Konfigurationsfile (**`config.lua`**):

```
configuration =
{
    url = „http://www.bbv.ch“,
    logfile = „/var/log/app.log“,
```


Ein minimales Programm, das die obige Konfiguration liest und verarbeitet, könnte wie folgt aussehen:

```
static void config_read(lua_State * lua,
        const char * key, std::string & value)
{
    lua_getglobal(lua, „configuration“);
    lua_pushstring(lua, key);
    lua_gettable(lua, -2);
    value = luaL_checkstring(lua, -1);
    lua_pop(lua, 1);
}

int main()
{
    lua_State * lua = luaL_newstate();
    luaL_openlibs(lua);
    luaL_dofile(lua, „config.lua“);
    std::string url;
    config_read(lua, „url“, url);
    lua_close(lua);
    printf(„url: ’%s’\n“, url.c_str());
    return EXIT_SUCCESS;
```

Natürlich hat ein reales Projekt deutlich mehr Konfigurationsparameter. So würde auch das Handling der einzelnen Parameter mit einem besseren Softwaredesign einfacher werden, z. B. eine Klasse mit einem konkreteren Interface, die den Lua State als privates Attribut versteckt. Die Idee und der grundlegende Mechanismus entsprechen aber im Wesentlichen dem obigen Beispiel.

5.2 EINHEITENUMRECHNUNG

Bei Systemen, die international zum Einsatz kommen, gibt es oft das Problem der Masseinheiten, z. B. bei deren Darstellung auf einem Display, Webinterface oder Ähnlichem. Die Schwierigkeit dabei sind länder- oder domänenspezifische Einheiten, die es darzustellen gilt. Oft soll auch eine benutzerdefinierte Einheit unterstützt werden.

Da zur Entwicklungszeit nicht alle Anforderungen bekannt sind, kann Lua helfen, solche Umrechnungen zu vereinfachen. Eine flexible Verarbeitung von Einheiten kann auch in C++ umgesetzt werden, hat jedoch die Konsequenz eines komplexeren Designs und potenziellen zusätzlichen Softwarereleases wegen neuer Anforderungen. So kann es vorkommen, dass aufgrund einer solch trivialen Anforderungsänderung ein neuer Release generiert werden muss, mit möglicherweise erheblichen Aufwänden.

Folgend nun ein simples Beispiel, wie mit Hilfe von Lua eine grosse Flexibilität darin erreicht wird, Einheiten umzurechnen.


```
#include <lua.hpp>
#include <string>
#include <cstdio>
#include <cstdlib>

double convert_temperature(
        double value, const std::string & conv)
{
    lua_State * lua = luaL_newstate();
    luaopen_math(lua);
    lua_pushnumber(lua, value);
    lua_setglobal(lua, „value“);
    luaL_dostring(lua, conv.c_str());
    double result = lua_tonumber(lua, -1);
    lua_close(lua);
    return result;
}

int main()
{
    std::string conv = „return value * 1.8 + 32“;
    double temperature = 20.0;
    printf(„Celcius : %f\n“, temperature);
    printf(„Fahrenheit: %f\n“,
        convert_temperature(
            temperature, conv));
    return EXIT_SUCCESS;
```

Der String zur Umrechnung **conv** muss nicht hardcodiert sein, möglicherweise stammt er aus einem Konfigurationsfile. Dies ist

der Applikation überlassen. Es ist auch denkbar, dass solche tiefen Eingriffe in das System lediglich von Experten, nicht jedoch vom Endanwender durchgeführt werden. Gerade auch in solchen Fällen ist eine hohe Flexibilität wünschenswert.

Zu beachten ist auch, dass der verwendete Lua State lediglich die math Library kennt und somit keine Möglichkeit hat, die Applikation zu beeinflussen.

5.3 ABLAUFSTEUERUNG

Als Beispiel dient hier eine Temperaturregelung. Die Applikation, geschrieben in C++, kann eine Heizung ein- und ausschalten sowie über einen Sensor die aktuelle Temperatur erfassen. Die eigentliche Steuerung soll nun aber in Lua umgesetzt werden, um bei einer Anpassung der Regelung die Applikation nicht neu kompilieren zu müssen.

So könnte auch ein sehr einfaches Software-Update bei Feldanlagen ermöglicht werden: Die Grundapplikation stellt den Zugang zur Hardware zur Verfügung und muss nicht geändert werden. Die Temperaturregelung ist in diesem Fall sehr einfach änderbar.

Das hier gezeigte Beispiel ist objektorientiert, und es wird SWIG verwendet, um das Interface zwischen C++ und Lua herzustellen. Es werden an dieser Stelle auch nur die Klassendeklarationen aufgezeigt.

Die Klassen `Heater` (Heizung) und `Sensor` (Temperatursensor) ermöglichen den Zugang zur Hardware.


```
class Heater
{
    public:
        void turn_on();
        void turn_off();
};

class Sensor
{
    public:
    double measure();
```

Das SWIG-Interface ist für diesen Fall trivial. Es werden lediglich die obigen Klassen benötigt. Zusätzlich wird eine Funktion nach Lua exportiert, die eine gewisse Anzahl Minuten wartet: **sleep_minutes**(...).

```
%module tempcontrol
%{
#include „Heater.hpp"
#include „Sensor.hpp"
extern void sleep_minutes(int);
%}
%include „Heater.hpp"
%include „Sensor.hpp"
extern void sleep_minutes(int);
```

Die Temperaturregelung (hier eine einfache Hysterese) könnte dann in Lua so aussehen:

```
heater = tempcontrol.Heater()
sensor = tempcontrol.Sensor()
while true do
    local t = sensor:measure()
    if t < 18 then
        heater:turn_on()
    elseif t > 25 then
        heater:turn_off()
    end
    sleep_minutes(15)
end
```

In dieser Schleife wird periodisch alle 15 Minuten die Temperatur gemessen und innerhalb der Grenzen von 18 bis 25 Grad Celsius gehalten.

6 FAZIT

Richtig eingesetzt können eingebettete Scriptsprachen viele Vorteile bieten. Sehr wichtig dabei ist das Design des Interfaces und die klare Trennung von Aufgaben. Es muss auch unbedingt darauf geachtet werden, wer welche Daten verwaltet. Falls ein komplexeres Interface von mehr als einem Dutzend Funktionen nötig ist, wird der Einsatz eines Werkzeugs wie SWIG empfohlen.

Ob für den konkreten Fall eine eingebettete Scriptsprache den Anforderungen genügt, bleibt abzuklären. Dabei sind auch die verfügbaren Ressourcen zu beachten. Abhängig von der Zielplattform ist der Einsatz eines just in time compilers (8) denkbar.

Das Vorgehen kann auch so aussehen, dass zu Beginn möglichst viel in Lua implementiert wird. Anschliessend überall dort, wo Ressourcen zu Einschränkungen führen, mit einer performanteren Implementation in C++ ersetzen. So nähert sich eine Software dem Optimum von Entwicklungszeit und Performance. Auf jeden Fall ist es empfehlenswert, eine eingebettete Scriptsprache in Erwägung zu ziehen.

7 ANHANG

7.1 AUTOR

Mario Konrad

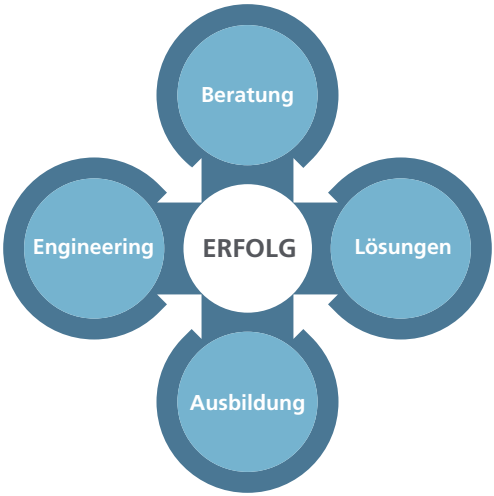
ist Dipl. Inf. Ing. HTL und arbeitete bei der bbv Software Services AG als Software-Architekt. Als Spezialist für Embedded Systeme hat er schon in einigen Projekten Lua als Embedded Scriptsprache erfolgreich eingesetzt.

7.2 QUELLENVERZEICHNIS

1. Lua Homepage. [Online] http://www.lua.org/.
2. Lua Users. [Online] https://sites.google.com/site/marbux/home/where-lua-is-used.
3. PUC-Rio. [Online] http://www.puc-rio.br/.
4. Lua Lizenz. [Online] http://www.lua.org/license.html.
5. Lua Referenz Manual. [Online] http://www.lua.org/manual.
6. Liste von Lua Bindings zu C++. [Online] http://lua-users.org/wiki/BindingCodeToLua.
7. SWIG Homepage. [Online] http://swig.org/.
8. Lua JIT. [Online] http://luajit.org/.

bbv
MAKING VISIONS WORK.

bbv Software Services AG ist ein Schweizer Software- und Beratungsunternehmen, das Kunden bei der Realisierung ihrer Visionen und Projekte unterstützt. Wir entwickeln individuelle Softwarelösungen und begleiten Kunden mit fundierter Beratung, erstklassigem Software Engineering und langjähriger Branchenerfahrung auf dem Weg zur erfolgreichen Lösung.



Unsere Booklets und vieles mehr finden Sie unter
www.bbv.ch/publikationen

MAKING VISIONS WORK.

www.bbv.ch · info@bbv.ch