Day 6
Extension and Scripting Languages

Joel Isaacson
Ascender Technologies Ltd.
http://ascender.com

Today's Lectures

- Today we will examine a number of extension and scripting languages.
- We will explore two low end languages, Tcl/Tk and Lua, that are suitable for embedded systems.
- Most of the techniques discussed are applicable to the “larger” languages such as Perl, Python, Ruby etc.

Tcl/Tk Why?

- It might not be apparent but extension languages are very useful for embedded systems.
- We will begin with some of the basics of Tcl.
- Tcl stands for Tool Command Language.
- Tk is its associated graphical user interface toolkit.

Basic Syntax of Tcl

http://www.bin-co.com/tcl/tutorial/intro.php

- Tcl/Tk is an easy language to master, it is not so easy to learn. It has a syntax which is different than most computer languages. Its syntax is closest to shell scripts:

```
a=3+4
```

- In Tcl this command is:

```
set a [expr 3 + 4]
```

Variables

- Tcl allows you to store values in variables and use the values later in commands. The set command is used to write and read variables.

```
set x 32
```

- To use the value of a variable in a command, '$' is pretended to the name

```
expr $x*3
```

Variables

- You can use variable substitution in any word of any command, or even multiple times within a word:

```
set cmd expr
set x 11
$cmd $x*$x
```

Command substitution

- You can also use the result of one command in an argument of another command. This is called command substitution:

```
set a 44
set b [expr $a*4]
```

Command substitution

- When a '[' appears in a command, Tcl treats everything between it and the matching ']' as a nested Tcl command. Tcl evaluates the nested command and substitutes its result into the enclosing command in place of the bracketed text. In the example above the second argument of the second set command will be 176.

Quotes and braces

- Double-quotes allow you to specify words that contain spaces. For example, consider the following script:

```
set x 24
set y 18
set z "$x + $y is [expr $x + $y]"
```

- After these three commands are evaluated variable z will have the value **24 + 18 is 42.**

Hello World

- Couldn't be easier

```
#!/usr/bin/tclsh
puts "Hello World"
```

Double Quotes

- Note that the double quotes (“) groups many strings to be one string.
- The Tcl interpreter will substitute variables ('$') and expressions ([]) within the doubly quoted string.

Some Basic String Manipulation Techniques

- The 'string' function is the basis for almost all string manipulation techniques.
- Please note that in all the cases, the result is returned.

```
set len [string length foobar]
=> 6
```

Some Basic String Manipulation Techniques

```
string index "See Spot run." 5
=> p
string range "See Spot run." 5 8
=> Spot
string range "See Spot run." 5
  end
=> Spot run.
```

Seaching and comparison

- We can search with the “string” function

```
set st "The trains were thirty minutes late
  this past week"

string first th $st

=> 16

string last th $st

=> 36
```

Length, case conversion, and trimming

```
string length "not too long"
=> 12
string toupper "Hello, World!"
=> HELLO, WORLD!
string tolower "You are lucky winner 13!"
=> you are lucky winner 13!
string trim abracadabra abr
=> cad
```

Lists

- Under Tcl, the value of each variable is stored as a string. Even if you want to save a number in a variable, this number is transformed into a string.
- As a special type of string, the list deserve a special attention in data representation in Tcl. The list is nothing more than a string with, as elements separator, the space.

Lists

```
set list {12 {78 5} 45 "Im a not a number"}
=> 12 {78 5} 45 "Im a not a number"
set sublist1 [lindex $list 1]
=> 78 5
set sublist2 [lindex $list 3]
=> Im a not a number
lindex $sublist2 2
=> not
```

How to display something?

To display a string, you can use the command 'puts'

```
% set variable 255

% puts "The number $variable"

The number 255

% puts [format "The number %d is equal to 0x%02X" $variable $variable]

The number 255 is equal to 0xFF
```

Control Flow

- The following commands are similar to the C equivalent.

```
if {...cond...} {...body...}

while {...cond...} {...body...}

for {... init ...} {...cond...} {...inc...}
 {...body...}

foreach varnames {...list...} {...body...}
```

Note: the '...cond...' is evaluated in the same way that it should be with command 'expr'.

While

```
set i 0
while {$i<4} {
 puts "$i*$i is [expr $i*$i]"
 incr i
 }
0*0 is 0
1*1 is 1
2*2 is 4
3*3 is 9
```

for

```
for {set i 0} {$i<4} {incr i} {
 puts "$i*$i is [expr $i*$i]"
 }
0*0 is 0
1*1 is 1
2*2 is 4
3*3 is 9
```

foreach

```
set observations \
  {Bruxelles 15 22 London 12 19 Paris 18 27}
 foreach {town Tmin Tmax} $observations {
 set Tavg [expr ($Tmin+$Tmax)/2.0]
 puts "$town $Tavg"
}
```

Bruxelles 18.5

London 15.5

Paris 22.5

Arrays

- Arrays are always unidimensional but the index is a string.
- If you use a separator in the index string (such as ',', '-'), you can get the same effect than with a multidimensional array in other languages.

Arrays

```
set observations \
  {Bruxelles 15 22 London 12 19 Paris 18 27}
foreach {town Tmin Tmax} $observations {
set obs($town-min) $Tmin
set obs($town-max) $Tmax
}
parray obs
```

```
obs(Bruxelles-max) = 22
obs(Bruxelles-min) = 15
obs(London-max)    = 19
obs(London-min)    = 12
obs(Paris-max)     = 27
obs(Paris-min)     = 18
```

Procedures

- Procedures are the equivalent of the C functions.

```
proc sum2 {a b} { return [expr $a+$b] }
```

- The default return value is the return value of the last evaluated function in this procedure. So the following script is perfectly equivalent :

```
proc sum2 {a b} { [expr $a+$b] }
```

Procedures

- The special argument name 'args' contains a list with the rest of the arguments.

```
proc sum {args} {
   set result 0
   foreach n $args {
      set result [expr $result+$n] }
   return $result }
sum 12 9 6 4
31
```

Procedures

- It is also possible to specify default parameters.

```
proc count {start end {step 1}} {
    for {set i $start} {$i<=$end} {incr i $step}
  {
    puts $i
  }
}
```

```
count 1 3
1
2
3
```

```
count 1 5 2
1
3
5
```

Global Variables

```
set global_counter 3
proc incr_counter {} {
    global global_counter
    incr global_counter
}
incr_counter
4
```

Eval

- The 'eval' command
  - concatenate all its arguments in one string
  - splits this string using spaces as separators
  - evaluate the command sentence formed by all the substrings
- In the following example, I used the function 'sum' that we have already defined.

Eval

```
proc average {args} {
  expr [eval sum $args] / [llength $args]]
}
average 45.0 65.0 78.0 55.0
60.75
```

Hello World

- Let us begin, as all other tutorials begin, with the "Hello World" program. Let's also make the program graphical with Tk.

```
#!/usr/bin/wish
label .hello -text "Hello World"
pack .hello
```

Label with value

- Let's change the label being displayed to an evaluated value
- Let's also change the font in the label

```
#!/usr/bin/wish
set a 8
label .hello -text “a[expr 4*$a]b” \
    -font "times 32"
pack .hello
```

Tk Button

- The button widget is like the label widget that has a callback

```
button .hello -text "Hello World"
  -command {puts "Hello World" ;
  exit }

pack .hello
```

Lua

- "Lua" (pronounced LOO-ah) means "Moon" in Portuguese.
- Lua is a powerful, fast, light-weight, embeddable scripting language.
- Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics.

Lua

- Lua is dynamically typed, runs by interpreting bytecode for a register-based virtual machine, and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and rapid prototyping.

Lua is a proven, robust language

- Lua has been used in many industrial applications (e.g., Adobe's Photoshop Lightroom), with an emphasis on embedded systems and games (e.g., World of Warcraft).
- Lua has a solid reference manual and there are several books about it.

Lua is fast

- Lua has a deserved reputation for performance.
- Lua is fast not only in fine-tuned benchmark programs, but in real life too.
- A substantial fraction of large applications have been written in Lua.

Lua is portable

- Lua is distributed in a small package and builds out-of-the-box in all platforms that have an ANSI/ISO C compiler.
- Lua runs on all flavors of Unix and Windows, and also on mobile devices and embedded microprocessors (such as ARM and Rabbit) for applications like Lego MindStorms.

Lua is embeddable

- Lua is a fast language engine with small footprint that you can embed easily into your application.
- It is easy to extend Lua with libraries written in other languages.
- It is also easy to extend programs written in other languages with Lua.

Lua is powerful (but simple)

- A fundamental concept in the design of Lua is to provide meta-mechanisms for implementing features, instead of providing a host of features directly in the language.
- Lua's meta-mechanisms bring an economy of concepts and keep the language small.

Lua is small

- Adding Lua to an application does not bloat it.
- The source contains around 17000 lines of C.
- Under Linux, the Lua interpreter built with all standard Lua libraries takes 153K.

Lua is free

- Lua is free software, distributed under a very liberal license (the well-known MIT license).
- It can be used for any purpose, including commercial purposes, at absolutely no cost.
- Just download it and use it.

Lua Data Types

- Number
- String
- Boolean
- Table
- Function
- Nil
- Userdata
- Thread

Numbers

- Lua allows simple arithmetic on numbers using the usual operators to add, subtract, multiply and divide.

```
> print(2+2)
4
> print(2-7)
-5
> print(7*8)
56
> print(7/8)
0.875
```

Strings

- Lua also uses strings (i.e. text) types:

```
> print("hello")
hello
```

- We can assign strings to variables just like we can numbers:

```
> who = "Lua user"
> print(who)
Lua user
```

Strings

- We can concatenate (join together) strings together using the .. operator between two strings.

```
> print("hello ")

hello

> print("hello " .. who) -- the variable
  "who" was assigned above

hello Lua user

> print(who)

Lua user
```

Boolean

- Boolean values have either the value true or false. If a value is not true, it must be false and vice versa.

```
> x = true
> print(x)
true
> print(not x)
false
> print(not false)
true
```

Boolean

- Boolean values are used to represent the results of logic tests.

```
> print(1 == 0)
false
> print(1 == 1)
true
> print(1 ~= 0)
true
> print(true ~= false)
true
```

Tables

- Lua has a general-purpose aggregate datatype called a table.
- Aggregate data types are used for storing collections (such as lists, sets, arrays, and associative arrays) containing other objects (including numbers, strings, or even other aggregates).

Tables

- Lua is a unique language in that tables are used for representing most all other aggregate types.
- Tables are created using a pair of curly brackets {} . Let's create an empty table:

```
> x = {}
> print(x)
table: 0035C910
```

Tables

- We can construct tables containing other objects, such as the numbers and strings described above, e.g.

```
> x = { value = 123, text = "hello" }
> print(x.value)
123
> print(x.text)
hello
```

Tables

- We can print the values out using the notation: table.item. We can also put tables inside other tables.

```
> y = { const={ name="Pi", value=3.1415927 },
  const2={ name="light speed", value=3e8 } }

> print(y.const.name)

Pi

> print(y.const2.value)

300000000
```

Functions

- In Lua, functions are assigned to variables, just like numbers and strings.
- Functions are created using the function keyword.
- We will create a simple function which will print a friendly message.

Functions

- `> function foo() print("hello") end -- declare the function`
- `> foo() -- call the function`
- `hello`
- `> print(foo) -- get the value of the variable "foo"`
- `function: 0035D6E8`

Functions

- Being a value just like any other, we should be able to assign functions to variables, just like the other values, and we can.

```
> x = function() print("hello") end

> x()

hello

> print(x)

function: 0035EA20
```

Nil

- nil is a special value which indicates no value. If a variable has the value nil then it has no value assigned to it.

```
> x = 2.5
> print(x)
2.5
> x = nil
> print(x)
nil
```

Userdata

- Userdata values are objects foreign to Lua, such as objects implemented in C.
- These typically come about when an object in a C library is exposed to Lua.
- An example of a userdata value is a file handle.

Threads

- A thread value represents an independent (cooperative) thread of execution.

Dynamic Typing

- You might have noticed that while we created the above variables, we did not have to specify which type of variable we were creating.
- For example,

```
a = 1

b = "hello"

c = { item1="abc" }
```

Querying type

- As Lua is a reflective language, we can use the Lua function type() to get athe type of a particular object.

```
> x = "123"  -- a string

> print(x, type(x)) -- show the value of x and its type

123     string

> x = x + 7  -- Notice the type change

> print(x, type(x)) -- again show the value and type

130     number
```

Multiple Assignments

- In Lua we can can perform multiple assignments in a single statement, e.g.,

```
> x, y = 2, "there"
> print(x,y)
2       there
```

- The list of values on the right is assigned to the list of variables on the left of the =.

Multiple Assignments

- We can assign as many values as we like and they don't all have to be of the same type. e.g.,

```
> a,b,c,d,e,f = 1,"two",3,3.14159,"foo",
  { this="a table" }

> print(a,b,c,d,e,f)

1  two  3  3.14159 foo  table: 0035BED8
```

Evaluation occurs before assignment

- Any expressions are evaluated first. The evaluated expression is then assigned.

```
> i = 7
> i, x = i+1, i
> print(i, x)
8       7
```

Swapping Values

- You can use multiple assignment to swap variable values around.

```
> a,b = 1,2  -- set initial values
> print(a,b)
1       2
> a,b = b,a  -- swap values around
> print(a,b)
2       1
```

Math

- We'll try a few of the functions and variables as an example.

```
> = math.sqrt(101)
10.049875621121
> = math.pi
3.1415926535898
> = math.sin( math.pi/3 )
0.86602540378444
```

Coercion

- Lua will automatically convert string and number types to the correct format in order to perform calculations.
- If the string cannot be converted to a number an error is raised.
- This automatic conversion of types is called coercion.

Coercion

```
> = 100 + "7"

107

> = "1000" + 234

1234

> = "hello" + 234

stdin:1: attempt to perform arithmetic on a string
   value

stack traceback:

        stdin:1: in main chunk

        [C]: ?

> = 234 + "1000"

1234
```

Tables as dictionaries

- Tables can also be used to store information which is not indexed numerically, or sequentially, as with arrays.
- These storage types are sometimes called dictionaries, associative arrays, hashes, or mapping types.

Tables as dictionaries

```
> t = { apple="green", orange="orange",
  banana="yellow" }

> for k,v in pairs(t) do print(k,v) end

apple   green

orange  orange

banana  yellow
```

Notes about table keys

- Lua stores all elements in tables generically as key-value pairs.
- In an array the keys are just number object keys.

```
> t = { 3,6,9 }  -- is the same as...
> t = { [1]=3, [2]=6, [3]=9 } -- is the same as...
> t = {} t[1]=3 t[2]=6 t[3]=9
```

Control Structure

- Lua control structures will be familiar to programmers.
  - while
  - repeat
  - for
  - if ... then ... else ... end

while

- The conditional looping statement while has the form:

```
while exp do block end
```

- For example, a simple loop:

```
> i = 3
> while i>0 do
>>  print(i)
>>  i = i-1
>> end
```

```
3
2
1
```

repeat

- The conditional looping statement repeat has the form:

```
repeat block until exp
```

- For example, a simple loop:

```
> i = 3
> repeat
>>   print(i)
>>   i = i-1
>> until i==0
```

```
3
2
1
```

if ... then ... else ... end

- The statement if has the form:

```
if 10>2 then print("bigger") end

bigger
```

- if ... then ... else ... end

```
> if 1>10 then print("bigger") else
  print("smaller") end

smaller
```

API for Tcl/C Integration

- You can embed both C in Tcl and Tcl in C.
-