Effective Scripting in Embedded Devices

Steve Bennett
WorkWare Systems
http://www.workware.net.au/
steveb@workware.net.au

April 2010

ABSTRACT

Scripting can be a valuable tool for embedded systems, but when is it most appropriate and which scripting languages are best under what circumstances?

This paper provides answers to these questions, along with case studies of situations where scripting has been used effectively in embedded systems.

1. INTRODUCTION

Anyone who has written a short shell script to rename some files, send a signal to a running process or to extract and reformat some messages from an apache log file knows that there are tasks which are far more easily achieved with a scripting language than with C or C++. Many embedded systems can benefit from the use of one or more scripting languages, even those that are relatively slow, have limited resources or no-MMU.

Creating an application in an embedded system generally involves accepting more tradeoffs and compromises than creating an application for a server environment, which is typically larger, faster and less mobile. As an embedded developer, it is always necessary to design a total solution to match the constraints of the environment, such as CPU speed, memory, flash, reliability, response time requirements, and cost.

Different scripting languages have different strengths and weaknesses—possibly more so than compiled languages since most compiled languages strive to be general-purpose languages, whilst many scripting languages do not. By choosing an appropriate scripting language and using it in appropriate ways, building an embedded product can be significantly faster and simpler.

2. WHAT IS EMBEDDED ANYWAY?

Or, Size Really Does Matter

Our company works with a wide variety of embedded platforms. Some developers consider a Core 2 Duo with 1GB of RAM to be "embedded" if it has a microATX form factor with no screen and keyboard, but that is a far cry from the typical embedded platform we work with.

Here are some fairly typical embedded platforms.

1. IXP425 533Mhz, 128M RAM, 32M Flash.
2. Coldfire m5282 66Mhz, 16M RAM, 4M Flash, SD card.
3. Microblaze soft CPU 100MHz, 32M RAM, 16M Flash.

For these systems, an application that is 10MB in size is going to take a fairly substantial amount of RAM and flash. It is possible that this is acceptable for the primary application—the raison d'être of the device—but if the application simply provides an ancillary management function, then likely not.

Consider a common requirement for these embedded devices; SNMP support. The simplest and most common approach is to use net-snmp. On Linux, this is a simple matter of compiling and installing. So what is the size of the installed application? Here are the sizes from a typical device.

```
-rwxr-xr-x 1 563688 14:35 /lib/libnetsnmp.so.5
-rwxr-xr-x 1 268380 14:35 /lib/libnetsnmpagent.so.5
-rwxr-xr-x 1 121104 14:35 /lib/libnetsnmphelpers.so.5
-rwxr-xr-x 1 446488 14:35 /lib/libnetsnmpmibs.so.5
-rwxr-xr-x 1  22900 14:35 /bin/snmpd
```

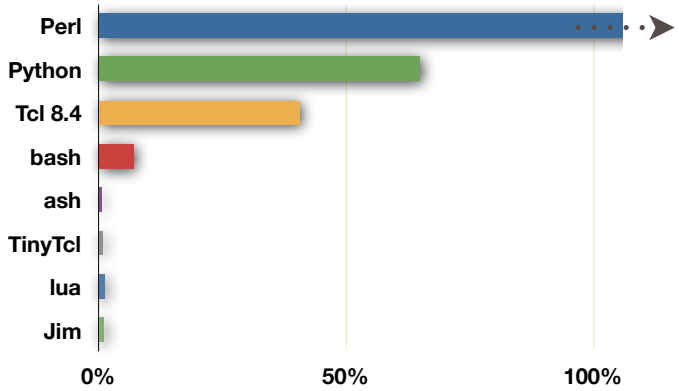
This is 1.3MB just for SNMP support. While this doesn't seem like much on a 50GB hard disk, or even a 1GB SD card, it quickly adds up for a device with only 8MB of flash.

Once a number of other common components are added, the system has a moderately large footprint even before any product-specific components have been added.

```
openssh server      241K
libssl/libcrypto   1177K
bash                513K
iptables            115K
strace              250K
tcpdump             247K
```

Larger applications means more RAM, more flash, larger upgrade images and increased application load times.

In the context of choosing a scripting language for an embedded system, size is one of the most important factors. The following graphs gives a rough estimate of the size of several popular scripting languages as a percentage of space available for applications on a device with 8M of flash[1].



[1] Includes only minimal language core. Excludes non-core packages, modules. Assumes 50% compression (e.g. squashfs)

3. SIZE AND SPEED ARE RELATED

Consider the following simple test:

```
puts "hello world"
```

hello.tcl

```
$ time tclsh hello.tcl
real    0m0.270s
```

Intel(R) Core(TM)2 Duo CPU @ 2GHz, 2GB RAM

Not too bad. 270ms.

```
$ time tclsh hello.tcl
real    0m0.043s
```

Intel(R) Core(TM)2 Quad CPU @ 2.33GHz, 4GB RAM

Much better. Only 43ms on this faster system.

```
$ time jimsh hello.tcl
real    0m 0.03s
```

XScale-IXP42x (v5b) @ 533MHz, 128MB RAM

Now we're getting somewhere! An embedded system can do it in only 3ms running Jim Tcl.

```
$ time jimsh hello.tcl
real    0m 0.01s
```

XScale-IXP42x (v5b) @ 266MHz, 32MB RAM

This slower system managed it in 1ms!

Why is this so? The big systems are running recent versions of Tcl, 8.4 or 8.5. These have lots of capabilities, but even starting the interpreter requires loading large binaries and/or shared libraries and parsing many initialisation files. The embedded systems are running a tiny version of Tcl. Around 150KB. The time to load and initialise the interpreter can be significant.

Now it can be argued that there are many reasons why these results can't be used directly.

Once the binaries and libraries are in the page cache, they will run much faster on subsequent invocations.

The time to load from disk is significant compared to running from flash or ramdisk.

However experience has shown, that in real-world use, a small application that loads quickly and does small job is far faster and less resource-intensive than a larger application doing the same thing. Consider an application written in Perl, Python or Tcl 8.5 on a 266MHz ARM system with 32MB of RAM. You should not be surprised if the system runs out of space or the application runs too slowly.

Small systems can be very fast as long as they don't have to do much.

It is easy for the overheads to swamp the potential performance improvement of a more complex system.

For example, there is no doubt that uClibc is generally slower than glibc for a number of operations. However if your application comes into existence, has a small amount of work to do (say, a handful of system calls) and disappears again, you will likely get better performance with uClibc than glibc.

4. BUILDING A REAL EMBEDDED PRODUCT

Our company spends a significant amount of time helping customers build and deploy real products. Almost all of these products run embedded Linux because of the fantastic leverage to be gained from a flexible kernel ported (and portable) to many platforms, and a large set of open source applications that are available with minimal work.

However, from the point of view of our customers, the most important aspect of the product is how it is differentiated, both in hardware and in software. This requires the development of custom applications for the product.

I often see two different approaches to building applications for embedded devices.

4.1 The embedded minimalists

These are the developers who will write everything in C from scratch to make it as small as possible. Every significant program contains a linked list implementation, a recursive descent parser, a configuration file parser and a TODO item, '***should use a hash table here but a fixed size array will do for now***'. No 3rd party libraries are used because the application contains proprietary IP and all the useful libraries are GPL. These projects spend lot of time tracking down crashes (I hope you have an MMU!) in newly written, poorly tested code.

Remember Greenspun's Tenth Rule [2]:

Any sufficiently complicated C program contains an ad-hoc, informally-specified, bug-ridden, slow implementation of half of a scripting language.

4.2 The application porters

On the other hand are the developers who don't really know anything about embedded systems. They just need to port their custom application from a Linux server (or Windows!) to the embedded system. So they start by trying to compile boost, php, postgresql and 10,000 lines of custom C++ which resists being cross compiled, assumes little-endian byte order and accesses fields in structures at odd byte boundaries. The questions to the mailing list usually start, '***I can't seem to get busybox to compile postgresql. I just get '***`elf2flt: ld.real not found`***'. Can someone please post the binaries***'.

Given the choice, I prefer the former approach to the latter. But does it need to be so hard? C is a great language for systems programming. Its ubiquity and maturity means that porting a well-written C application to an embedded target is often reasonably straightforward. But some of the core weaknesses of C are string handling, built-in data structures (i.e. none) and memory management. Yet, these are exactly the strengths of a typical scripting language.

Marrying a C application that does what it is good at (bit twiddling, efficient storage of data) with a scripting language can provide a best-of-both-worlds scenario.

5. CASE STUDY: AUTOMATED TESTING

Some years ago I worked at a company which produced a line of embedded Linux firewall-routers. As the number of models and the feature set of these devices grew, we realised the need for comprehensive automated system testing.

The initial approach was to have the automated test framework telnet to a device, set up various configurations and run various tests. This was scripted from a test system using Expect. While this generally worked, it was quite cumbersome as everything had to go through the telnet interface and relied on the limited set of command line tools available on the device. Even simple tasks, such as creating configuration files, were difficult through telnet. Some of these devices even had a shell without command-line redirection.

When we created the next version of the product, we added scripting support (TinyTcl) for manipulating the system configuration, including parsing and writing configuration files.

Suddenly we were able to discard the previous telnet-based automated testing approach and instead use a dynamic script-based approach. We did this by creating a tiny test script that ran under inetd on a certain port. This script accepted scripts as requests and executed them[2]. Since the script system had access to the configuration system, it was very easy to test different scenarios by making configuration changes. Also, the full power of the scripting language was available to execute commands and parse system logs and other files. We were able to create higher-level test components, all based on sending (small) scripts to be executed on the device.

A typical test script might look like:

```
source $testlib
use netconf net
test cable {
   # Find a dhcp connection we can use
   array set conn [netconf_find dhcp]
   # Configure it
   remote dev=$conn(dev) devname=$conn(devname) {
      config load -update
      set eth [config ref eth<devname=$dev>]
      set o [config new dhcp interface $eth]
      config set $o type cable
      if {$devname != "eth0"} {
         config set $o fwclass wan
      }
      config set $eth conn $o
      config save
   }
   # Wait for it to come up
   net_wait $conn(intf)
   pass "cable connection on $conn(intf) OK"
}
```

The `remote { ... }` command sets some remote variables from local variables and then sends the script to the device under test for execution.

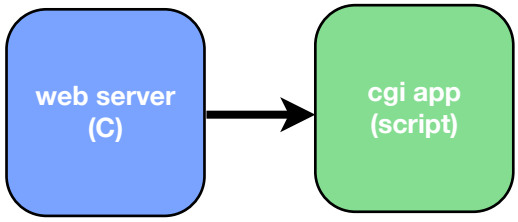
This approach allows for almost unlimited capability for the automated test system, as test scripts have access to the entire configuration system (via Tcl bindings) and to almost all aspects of the device (exec commands, signal processes, read and write files, etc.). The test scripts required far fewer changes as new features and new models were added. It was generally straightforward to abstract away differences between models in the test framework.

6. CASE STUDY: WEB FRAMEWORK

At WorkWare we have a product, µWeb, that makes it very easy to build web interfaces for embedded devices. The framework provides all the core functionality for implementing an embedded web application, but each product has its own requirements for what needs to be configured, what status should be displayed and the administrative actions that should be available.

Now a typical split for an embedded device would be between the web server and a cgi script or program that implements the web interface.

Here is how this looks:



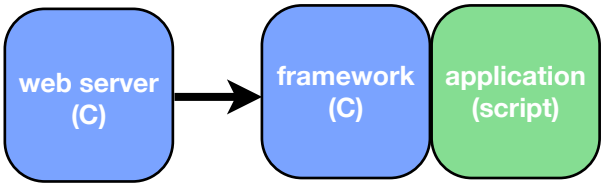
The web server's job is quite simple. It parses a network request, sets some environment variables, execs the cgi-bin app and sends the results back to the client. The cgi script/application is responsible for most of the work; parsing GET variables, POST variables (include multipart/form-data), cookies, managing session-based authentication, system state, validating input and generating html output.

Using a scripting language for the cgi "script" isn't a very good idea for performance reasons.

Consider this—we allow a 250ms budget for a web request. For any typical request that doesn't require generating a large amount of data, it should take no more than 250ms from when the user presses a button until the resulting web page is displayed—even on a slow device.

Implementing all of the cgi script functionality in a scripting language makes this a difficult (if not impossible) target to meet on a slower device.

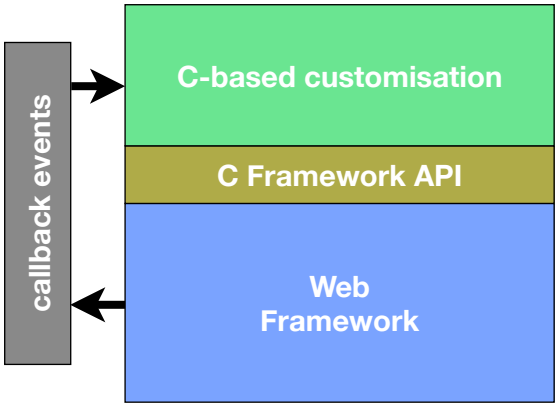
So instead, we split our framework like this:



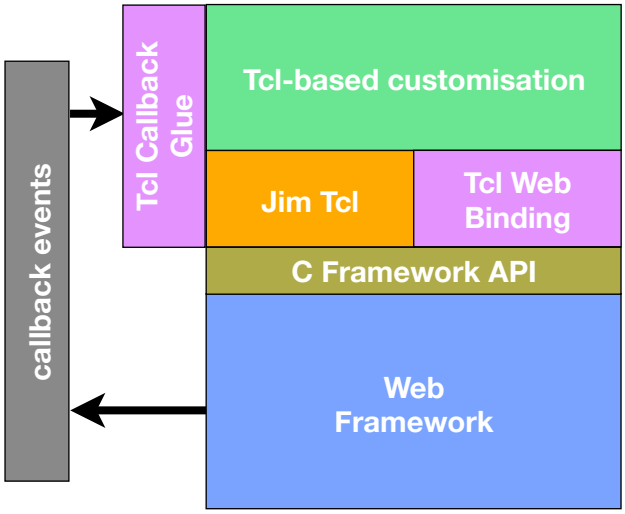
Here the framework provides protocol support (GET/POST/cookies/headers), authentication, state management, validation and error handling and layout, while the application script(s) provides the customisation.

Below is a more detailed representation of the framework/application architecture. Certain events (GET request, POST request, “display this field”) cause callbacks to be invoked, and those callbacks have access to the framework via a C-based API.

[2] Naturally this script only ran during automated testing to avoid a rather large security hole.



Now here the architecture is extended to support customisation via a scripting language, Jim Tcl [3], instead of via C-based callbacks [3].



In this case, application-specific functionality is implemented as Tcl ***scriptlets***. These are small scripts that are executed to provide the functionality for a single request.

When an event occurs, a thin Tcl callback layer causes the appropriate Tcl scriptlet to be invoked. The scriptlet has access to the framework API via a Tcl binding. It also has access to all the Tcl commands.

Here is a typical scriptlet.

```
submit -tcl {
   set zones [readfile $zonefile]
   writefile $tzfile $zones([cgi get tz])
   writefile [cgi configdir]/ntpserver \
      [cgi get ntpserver]
   catch {exec killall msntp}
}
```

All the core framework APIs are bound under a single command, `cgi`. It is straightforward to create the C-Tcl binding and, in general, the Tcl API is easier to use than the C API, mainly thanks to default arguments, untyped values and built-in lists and arrays/dictionaries.

With all the heavy lifting done by the framework, meeting overall performance requirements is generally quite easy. As mentioned earlier, the creation and initialisation of the interpreter needs to be very fast, on the order of 10ms or less.

Here is the timing for a typical request[4]:

round trip latency	38ms
interpreter creation	4ms
POST scriptlet	17ms
display scriptlet	2ms
framework processing	12ms
Total response	73ms

And similarly when handling a request completely in C:

round trip latency	38ms
POST scriptlet	1ms
display scriptlet	2ms
framework processing	12ms
Total response	53ms

Notice that, although the time to create the interpreter and run the script is 21 times longer than for the C version, the total response time is not significantly different and the total response time is well below 250ms where the system may appear sluggish.

On the other hand, implementing the Tcl version of the script is far easier than implementing the C version.

6.1 What can an extension script do?

Unlike an extension written in C, a script-based extension does not have unfettered access to libc and system calls. So what can a script do?

Firstly, the script must be able to access and manipulate application objects and state. In the case of µWeb, this means Tcl access to the C-based extension API.

Secondly, the script uses language features such as lists, string manipulation and flow-of-control commands.

Thirdly, the script must be able to interact with the system. This means:

- Reading and writing files (configuration, `/proc`, `/sys`, etc.)
- Examining filesystem state (`glob`, `file`)
- Running commands (`exec`)
- Parsing files and command output (`regexp`, `regsub`, `string`)
- Sending signals to processes (`kill`)

3 The real framework allows any callback to be implemented in either C or Tcl. This allows omitting the scripting language entirely if space is at a premium, or allows certain functionality to use C where this makes system interfacing simpler or high performance is required.

4 Timing tests were performed on an IXP420-based systems @ 266MHz

This third point is a significant difference from a scripting language embedded in a non-embedded application, where the script would typically have limited interaction with the system.

7. EMBEDDING JIM TCL

Integrating Jim Tcl into an application is quite straightforward. Jim Tcl is written in quite portable C and has a minimal autoconf-based configure system. Configuring, building and linking Jim won't be covered here. See [3] for more information.

7.1 Creating the interpreter

Generally at application startup (but maybe when first needed), the interpreter needs to be created.

```
#include <jim.h>

Jim_Interp *create_tcl_interp(void)
{
   const char *p;
   Jim_Obj *listObj;
   Jim_Interp *interp = Jim_CreateInterp();
   Jim_RegisterCoreCommands(interp);
   Jim_InitStaticExtensions(interp);
   /* Add TCLLIBPATH to JIM_LIBPATH */
   listObj = Jim_GetVariableStr(interp, JIM_LIBPATH,
      JIM_NONE);
   if (Jim_IsShared(listObj))
      listObj = Jim_DuplicateObj(interp, listObj);
   p = getenv("TCLLIBPATH");
   if (p) {
      const char *end;
      do {
         int len = -1;
         /* Allow either ' ' or ':' separators */
         end = strchr(p, ' ') ?: strchr(p, ':');
         if (end) len = end - p;
         Jim_ListAppendElement(interp, listObj,
            Jim_NewStringObj(interp, p, len));
         p = end + 1;
      } while (end);
   }
   /* And standard paths */
   Jim_ListAppendElement(interp, listObj,
      Jim_NewStringObj(interp, "/lib/jim", -1));
   Jim_ListAppendElement(interp, listObj,
      Jim_NewStringObj(interp, "/lib/tcl6", -1));
   Jim_SetVariableStr(interp, JIM_LIBPATH, listObj);
   return interp;
}
```

At this point, any application-specific extensions (commands) are registered, and any application-specific variables or procedures may also be created.

7.2 C to Tcl

When an event occurs that requires a Tcl script to be invoked, this may be done in one of two ways:

```
/* Evaluate a script. filename and line indicate the
original source location */
int
Jim_Eval_Named(Jim_Interp *interp,const char *script,
const char *filename, int line);

/* Evaluate a script from a file */
int
Jim_EvalFile(Jim_Interp *interp, const char *file);
```

If the script is embedded in the executable as a string, `Jim_Eval_Named()` can be used. If the script exists in the filesystem (e.g. a configuration file), `Jim_EvalFile()` can be used.

Context-specific variables can be set before invoking the script, and the return code and variable values may be used to retrieve a result (if required), in addition to any side effects of the script invoking application-specific APIs through a Tcl binding.

7.3 Tcl to C

In addition to interfacing with the system via standard I/O, process and signal commands, an extension script will generally need to access the application objects and commands through one or more application-specific commands.

Here is a simple example of creating a Tcl command, `sleep`.

```
static int Jim_SleepCmd(Jim_Interp *interp,
   int argc, Jim_Obj *const *argv)
{
   if (argc != 2) {
      Jim_WrongNumArgs(interp, 1, argv, "seconds");
      return JIM_ERR;
   }
   else {
      double t;
      int ret = Jim_GetDouble(interp, argv[1], &t);
      if (ret == JIM_OK) {
         if (t < 1)
            usleep(t * 1e6);
         else
            sleep(t);
      }
      return ret;
   }
}

/* to create the command ... */
Jim_CreateCommand(interp, "sleep", Jim_SleepCmd,
NULL, NULL);
```

For complex interfaces, Jim supports the creation of subcommands via a table/callback approach.

8. MAKING THE RIGHT CHOICE

There are many different scripting languages to choose from, but some choices are better than others for embedded systems.

What are we looking for in an embedded scripting language?

- Written in portable C
- Designed to be embedded, not standalone
- Small
- Fast to start
- Modular, to allow unneeded features to be removed
- BSD or equivalent licence

And as a bonus:

- Identical (or at reasonably similar) to a popular language to leverage existing skills.

8.1 Why Tcl? Why Jim?

Some years ago, I was looking for an embeddable scripting language that would help with parsing and writing configuration files in an embedded device.

Consider some of the features of Tcl that make it work well in this context:

- regular expressions (great for ad-hoc parsing)
- powerful exec command
- associative arrays and lists (for managing data)
- filesystem commands: file, glob, open, close, read, write

Tcl makes it easy to use the power of Tcl from C and the power of C from Tcl.

I also very much like the fact that Tcl can look like it is not a language at all.

```
interface eth0
maxsize 1024
listen 80 192.168.1.20:8080
```

By defining appropriate procedures, `interface`, `maxsize` and `listen`, it is possible to make a Tcl script look like a configuration file.

The same is true for providing an interactive interface. Procedures can be created for custom commands, providing an interactive console with essentially no work.

We ended up using TinyTcl[4]. This was a slightly modified version of Tcl 6.7, the last version of Tcl that was still focussed on being an embedded language rather than a full application development language[5].

TinyTcl was very small, and yet supported the useful system-interfacing features described above. We also used this language very successfully in the early version of µWeb.

However TinyTcl suffered from a number of flaws.

- Scripts are re-parsed on every iteration. This means that parsing a 2000 line text file in a loop can be noticeably slow.
- No support for 64 bit integers
- No support for strings containing nulls
- Arrays are not first-class objects, which means passing arrays by name or constantly flattening and unflattening arrays
- No support for functional programming such as lambdas
- Error reporting is poor
- No list expansion operator, `{*}`

Many of these are issues that Tcl as a whole has struggled with for many years, and some have only been addressed in the very latest (as yet unreleased) Tcl 8.6. Some of these required very significant changes to Tcl and back-porting them to TinyTcl was not feasible.

Fortunately, Salvatore Sanfilippo was particularly interested in functional programming with Tcl and created a from-scratch reimplementation of the Tcl language with a focus on addressing many of these issues. Going back to the roots of Tcl, Jim also focussed on embedding as a goal.

I considered using Jim as a replacement for TinyTcl, however it lacked some significant features from Tcl.

- regular expressions
- exec
- arrays (via the array command)
- compatibility with the Tcl I/O commands
- documentation

Also there were quite a number of bugs exposed by running existing Tcl scripts.

Over a period of time, I ported various features from TinyTcl and implemented others directly in Jim, including some features from Tcl 8.4, 8.5 and 8.6. After some time, we now have a small, embeddable language with a high level of compatibility with Tcl and a number of advanced features, including some not yet even available in Tcl.

This version of Jim is publicly available via git and the web site: http://jim.workware.net.au/. We are currently working to merge some or all of these changes into the Jim Tcl mainline.

8.2 Why (not) Lua?

Lua[5] is a very interesting language and has become quite popular. It was designed to fill the same need as Tcl originally had. That is, as an embeddable language in a host application.

Taken directly from the Lua web site:

- Lua is a proven, robust language
- Lua is fast
- Lua is portable
- Lua is embeddable
- Lua is powerful (but simple)
- Lua is small
- Lua is free

In addition, Lua can generate byte code that can be executed directly, which makes it interesting to be able to compile scripts on a host systems and embed the resulting byte code in the application.

For our application, µWeb, we decided against Lua for a few reasons:

- Built-in support for system interfacing is somewhat lacking (e.g. regular expressions, limited os library).
- The use of metatables and metamethods is interesting, but can be complex for non-programmers.
- Lua syntax isn't ideal for configuration files or interactive use.

---

5 Interestingly, I believe that the one thing which contributed most to the popularity of Tcl also brought about it's downfall, Tk. At the time, this was a revolutionary way to create a GUI. Compared to building a GUI with Xt or Motif, a Tcl/Tk-based GUI could be put together in no time and gradually evolve. However this encouraged people to create full, non-embedded applications with Tcl and send it on it's current path - where Tcl/Tk is no longer very popular for GUI applications and Tcl has been largely surpassed by languages such as Python, PHP and perhaps Ruby. On the other hand Tk has lived on as the standard UI for Python.

An early version of µWeb did have support for extensions written in Lua. It was quite easy to embed and worked well. Lua is in widespread use in both embedded and non-embedded application and it continues to grow and evolve.

Lua is probably the most significant alternative to Jim Tcl and is worth investigating.

8.3 Other Scripting Languages

There are many, many scripting languages available. Some of these that look interesting and may be suitable for embedded systems are:

- Pawn (formerly Small)
- Pike
- Nesla

9. LEVERAGING SCRIPTING

Once your embedded system contains a scripting language, it can make sense to leverage that support for small tasks.

There are a few ways that this can be done.

9.1 Testing support

We have built a product where one device is repurposed as a test and calibration jig. This device has identical hardware and software to the production device except it supports a simple menu system that allows technicians to run calibration and system tests on attached hardware.

These simple menu systems take user input, read files, write files, execute commands and display system state. They are also one-off and are modified as required. They are a perfect fit for a scripting language.

```
Vendor/product Version 1.0  Mar 19 12:23:35 EST 2010

  1.  Modem 1                    [Active]
  2.  Modem 2                    [Not Installed]
  k.  Modulation Control         [Running]
  t.  Modem Test Signal (0x1B)   [None (0)]
  m.  Modulation (0x01)          [QPSK (0x00)]
  q.  Quit

Select option []:
```

9.2 Prototyping

During the development of a system, it is often necessary to quickly bring up small components. For example, in one system we needed to wait for the hot-plugging of a modem board, check system configuration, program the FPGA appropriately and configure appropriate settings based on the current system status and configuration. It also needed to monitor system changes and reprogram the FPGA and/or change the system configuration as appropriate.

This job was too complex for a simple shell script, but it would have taken quite some time to create a robust C implementation.

Since we already had Tcl on the device, we were able to very quickly create a Tcl implementation of this component. Aspects of this implementation changed a number of times during the development of the product. However, once the system stabilised, we rewrote this component in C to reduce its memory footprint.

9.3 Replacing complex shell scripts

If a more capable scripting language is not available, it is possible to end up with a complex and slow shell script, which may run at a time-critical point, such as system boot.

In one system, the majority of the user configuration was stored in a text file that looked something like this:

```
net.ipaddr=10.0.0.200
net.subnet=24
snmp.community=public
snmp.location=Remote Office
```

Most parts of the system accessed this configuration through a C-based library; however, at system start-up, shell scripts needed access to this configuration. Parsing this configuration file in a shell script is cumbersome, but writing a small Tcl script to set environment variables based on the configuration is trivial.

This script, config-setenv, would produce:

```
config_net_ipaddr=10.0.0.200
config_net_subnet=24
config_snmp_community=public
config_snmp_location="Remote Office"
```

Using this from a shell script is as simple as:

```
eval `config-setenv`
ifconfig eth0 $config_net_ipaddr/$config_net_subnet
...etc...
```

Once you have access to a general-purpose scripting language, it becomes natural to solve these types of problems with it rather than resorting to either a complex combination of shell, awk, sed, tr, cut and sort, or otherwise writing it in C.

10. METHODS OF EMBEDDING SCRIPTS

In order to execute a script in the interpreter, the script source needs to be available. There are two obvious ways to do this when a scripting language is used as an extension language.

Load scripts from files at runtime

This simple approach uses some algorithm to determine the script(s) to load and execute. The algorithm is trivial when cgi applications are implemented completely in the scripting language.

This is also the approach that would be used to load scripts as configuration files.

Embed scripts within the application code

This is the approach we chose for µWeb. Here, each script is stored in an internal data structure (along with its original source location), and executed (evaluated) at the appropriate time.

We chose this approach in µWeb for a number of reasons.

- It allows a web application to be deployed as a single executable rather than an executable and a large collection of scripts.

- In µWeb, scripts are more correctly called 'scriptlets'. Each script is invoked within a certain context and does a small amount of work. It is simpler for the application developer to implement a "page" completely in a single source file rather than storing each script in a separate file.
- As a framework, development of a µWeb application already requires the user to build and link an executable. If µWeb were instead a fixed binary, more like a CAD application, it would make more sense to keep the user's scripts entirely separate from the application.

This is also the approach you would use for the popular ***template*** approach where HTML pages are interspersed with scripts.

11. MORE ABOUT JIM TCL

There are some differences between Jim and both TinyTcl and regular Tcl that are interesting to explore in more detail.

The expand operator {}*

One of the things that has been cumbersome in Tcl for many years is the inability to seamlessly convert between lists and procedure arguments. Consider this example:

```
# Return the largest value in the argument list
proc max {args} {
   set max [lindex $args 0]
   foreach v $args {
      if {$v > $max} {
         set max $v
      }
   } return $max
}
. max 5 10 7
10
```

Now what if we have a list of values? We would like to do something like:

```
. set l {1 2 3 4 5}
. max $l
1 2 3 4 5
```

This doesn't work too well. The problem is that the list needs to be expanded to multiple arguments. Typically this is solved in Tcl as follows:

```
. eval [concat max $l] 5
```

This is rather ugly. In many ways it is like the impedance mismatch in C with varargs, where two versions are needed for every function that takes variable arguments, such as `syslog()` and `vsyslog()`, `fprintf()` and `vfprintf()`.

So, Tcl 8.5 finally introduced the following list expansion syntax.

```
. max {*}$l
5
```

This one change made a huge difference in the usability of Tcl. Jim added support for the expand operator, `{*}`, in its earliest versions.

Isomorphic list-dictionary duality

In Tcl, everything is a string—except when it isn't: associative arrays are not first class objects, so they aren't strings. Tcl 8.5 introduced dictionaries to address this issue; however, arrays are still not first class objects.

Jim does this much better. Consider:

```
. set a {1 one 2 two 3 three}
. lindex $a 3 two
. puts $a(3) three
```

Note that the list with an even number of elements is "magically" transformed into an array as needed.

Compare with Tcl:

```
% puts $a(3)
can't read "a(3)": variable isn't array
```

No namespaces

Once Tcl grew to be used more for application development rather than as an embedded scripting language, the issue of name clashes became a problem. Tcl introduced namespaces, so now we have wonderful things like:

```
% ::fileutil::magic::filetype filename
```

While this is probably necessary for a full application development language, it just makes typical embedded usage more verbose.

Accurate runtime error messages with source location

One of the difficulties with a dynamic language is that a script being evaluated may not even **have** a source location, so reporting meaningful runtime error messages can be difficult.

Consider:

```
set a "puts"
append a " hello extra-arg"
eval $a
```

When the run-time error occurs, what file and line should be reported? There is none.

Nonetheless, in many situations it is possible to determine the source location. Jim carefully tracks a source location for each token in a script, in order to provide a meaningful error message.

Consider the following script, `test.tcl`:

```
proc a {} {
   b one two three
}
proc b {first args} {
   c $first
}
proc c {id} {
   expr {$id + 10}
}
a
```

First, when run under Tcl:

```
can't use non-numeric string as operand of "+"
    while executing
"expr {$id + 10}"
    (procedure "c" line 2) invoked from within
"c $first"
    (procedure "b" line 2) invoked from within
"b one two three"
    (procedure "a" line 2) invoked from within
"a"
    (file "test.tcl" line 11)
```

Now when run under Jim:

```
test.tcl:8: Runtime Error: expected number but got
"one" in procedure 'a' called at file "test.tcl",
line 11 in procedure 'b' called at file "test.tcl",
line 2 in procedure 'c' called at file "test.tcl",
line 5 at file "test.tcl", line 8
```

Firstly, the error message is far more compact with Jim. Since Jim is able to accurately report source locations, it doesn't need to include the context as Tcl does.

Secondly, we can immediately see that the error occurred on line 8 of `test.tcl`, whereas we have to go find procedure "`c`" and count source lines to find the location in Tcl.

This works especially well when scripts are embedded into C code. Consider the original source:

```
test.page:
....
   submit {
      puts "This is the submit script"
      error here
   }
...
```

If the original source line in test.page can be determined, it is possible to invoke `Jim_EvalNamed(script, "test.page", line)` to inform the interpreter of the original source location. This allows runtime errors to refer to the original source location.

This difference between Tcl and Jim has made Jim far easier to use in µWeb.

12. POTENTIAL ISSUES

There are some potential issues that should be considered when choosing and embedding a scripting language.

Stack Space

Many scripting languages make heavy use of the stack. This can be a problem for a platform with small or fixed size stacks.

Some scripting languages support keeping the language stack on the heap (sometimes known as NRE, or Non-Recursive Engine); however, interfacing those languages with C is more difficult since C wants to keep its state on the stack.

No-MMU Support

Some scripting languages fundamentally assume the existence of an MMU and `fork()` (Perl, Python, bash, busybox ash), while some have various levels of support for no-MMU systems.

- Jim Tcl provides a blocking-only `exec` command on no-MMU systems.
- busybox hush supports many, but not all, Bourne shell features on no-MMU systems.

Licensing

In an embedded device, the most useful application to embed a scripting language is likely to be a proprietary one. This means that a GPL or similar licensed scripting language is out of the question. Fortunately Tcl (including all versions of standard Tcl, as well as Jim Tcl) and Lua are available under a BSD-style licence.

13. CONCLUSION

Most developers appreciate that, when used well, scripting can provide a dramatic increase in productivity over traditional compiled languages (Witness the efficiency of Ruby on Rails compared to Java for certain applications, and the stampede of developers to it over the last few years.) The same is true for embedded systems, where a designed-for-embedded scripting language such as Jim Tcl or Lua can significantly increase productivity and reduce time to market.

14. ACKNOWLEDGMENTS

Thanks to Salvatore Sanfilippo for creating Jim.

Thanks to John Ousterhout for originally creating Tcl.

15. REFERENCES

[1] Bezroukov, N.. A Slightly Skeptical View on Scripting Languages. (http://www.softpanorama.org/People/Scripting_giants/scripting_languages_as_vhll.shtml)

[2] http://en.wikipedia.org/wiki/Greenspun's_Tenth_Rule

[3] http://jim.workware.net.au/

[4] http://tinytcl.sourceforge.net/

[5] http://www.lua.org/