Effective Scripting in Embedded Devices

Steve Bennett

WorkWare Systems
What is Embedded?
Creating an Embedded Product

- Time to market
- Quality
- Features
- Cost
- Size
- Performance

- Linux kernel
- uClibc
- Busybox
- Other open source
- Custom drivers

- Custom applications
Embedded Applications

Greenspun’s Tenth Rule

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

Embedded Minimalists
- linked list
- hash table
- exec wrapper
- config parser
- customisation API

Application Porters
- C++ toolchain
- Boost
- PostgreSQL
- byte order
- unaligned access
Language Strengths

C/C++
- Bit/byte twiddling
- Efficient storage
- Access entire system API
- Compiled code

Scripting
- String mangling
- Lists, Dictionaries
- Searching, Sorting
- Customisation
Make it Fit

Percentage of 8MB NOR flash used

- App space: 75%
- Kernel: 13%
- Config: 6%
- uClibc: 3%
- Boot loader: 3%
Make it work

Percentage of 8MB NOR flash used

- Kernel: 13%
- Config: 6%
- snmpd: 8%
- ssl/libcrypto: 7%
- bash: 3%
- iptables: 3%
- openssh: 1%
- uClibc: 3%
- strace: 2%
- openssh: 1%
- App space: 53%
- Boot loader: 3%
Add Scripting

Percentage of available space used by “core” scripting language

- Perl
- Python
- Tcl 8.4
- bash
- ash
- TinyTcl
- lua
- Jim
Languages Attributes

- **C**
- **C++**
- **Java**
- **ash**
- **bash**
- **Python**
- **Perl**
- **Jim**
- **Tcl**
- **Lua**

- Resource Efficient
- Slow Development
- Resource Hungry
- Rapid Development
Growth over Time

Minimal Linux Kernel

Note: Sizes are indicative only
Making big things small

• It is hard since all features are critical to someone

• Minimal Tcl - 5+ years with no progress

• Deeply Embedded Python - abandoned

• miniperl - unsupported

• Much easier to start small and focussed
All things being equal, large applications and libraries are slower to load and run than small applications and libraries.

<table>
<thead>
<tr>
<th>System</th>
<th>Time</th>
<th>System Calls</th>
<th>Relocations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel(R) Core(TM)2 Quad CPU 2.33GHz, 4GB RAM</td>
<td>43ms</td>
<td>173</td>
<td>3740</td>
</tr>
<tr>
<td><strong>Tcl 8.4 (glibc)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XScale-IXP42x (v5b) 266MHz, 32MB RAM</td>
<td>1ms</td>
<td>37</td>
<td>766</td>
</tr>
<tr>
<td><strong>Jim Tcl (uClIBC)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Simple ‘Hello World’ Test
Case Study
Automated Testing

Test Host
(Expect)

Device under Test

Tcl scripts

Telnet

TinyTcl Test Script

Expect + inetd + TinyTcl
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"
Case Study
Web Framework

Traditional
- web server (C)
- cgi app (script)

Embedded Scripting
- web server (C)
- framework (C)
- application (script)
μWeb
C-based Customisation

callback events

C-based customisation

C Framework API

Web Framework
submit -c {
    const char *tz = cgi_get("tz");
    /* find timezone spec for selected TZ */
    FILE *fh = fopen(ZONEFILE, "r");
    while ((fgets(buf, sizeof(buf), fh) != NULL) { 
        /* parse line,
         * match timezone,
         * write to /etc/TZ
         */
        ...
    }
    fclose(fh);

    /* write ntpserver */
    snprintf(buf, "%s/ntpserver", cgi_configdir());
    fh = fopen(buf, "w");
    fprintf(fh, "%s\n", cgi_get("ntpserver");
    fclose(fh);

    /* should use msntp.pid, ...*/
    system("killall msntp");
}
μWeb
Jim Tcl Scripting

Tcl Callback Glue

Tcl-based customisation

Jim Tcl
Tcl Web Binding

C Framework API

Web Framework

callback events
submit -tcl {
    # read timezones
    set zones [readfile $ZONEFILE]

    # write /etc/TZ
    writefile /etc/TZ $zones([cgi get tz])

    # write ntpserver
    writefile $CONFDIR/ntpserver [cgi get ntpserver]

    # kill (and respawn) msntp
    kill -TERM [readfile /var/run/msntp.pid]
}

What scriptlets do

• Access application API (Tcl commands)
• Examine/update strings, lists, arrays
• Use standard Tcl commands
• Interact with OS - files, commands, processes
How Fast is it?

C-based

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

In this case, application-specific functionality is implemented as Tcl scriptlets. These are small scripts which 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 [read file $zonefile]
    write file $tzfile $zones ([cgi get tz])
    write file [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:

<table>
<thead>
<tr>
<th>Description</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>round trip latency</td>
<td>38ms</td>
</tr>
<tr>
<td>POST scriptlet</td>
<td>1ms</td>
</tr>
<tr>
<td>display scriptlet</td>
<td>2ms</td>
</tr>
<tr>
<td>framework processing</td>
<td>12ms</td>
</tr>
<tr>
<td>Total response</td>
<td>53ms</td>
</tr>
</tbody>
</table>

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 Web, this means Tcl access to the C-based extension API.

Secondly, the script will use the language features such as list and string manipulation and flow-of-control commands.

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

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

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 in performance-critical situations.

Timing tests were performed on an IXP420-based systems @ 266MHz.
How Fast is it?
Tcl Scripting

Now here the architecture is extended to support customisation via a scripting language, Jim Tcl, instead of via C-based callbacks. In this case, application-specific functionality is implemented as Tcl scriptlets. These are small scripts which 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 [read file $zonefile]
    write file $tzfile $zones [cgi get tz]
    write file [cgi configdir] /ntp server [cgi get ntp server]
    catch {exec killall msn ntp}
}
```

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:

<table>
<thead>
<tr>
<th>Component</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>round trip latency</td>
<td>38</td>
</tr>
<tr>
<td>interpreter creation</td>
<td>4</td>
</tr>
<tr>
<td>POST scriptlet</td>
<td>17</td>
</tr>
<tr>
<td>display scriptlet</td>
<td>2</td>
</tr>
<tr>
<td>framework processing</td>
<td>12</td>
</tr>
<tr>
<td>Total response</td>
<td>73</td>
</tr>
</tbody>
</table>

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 Web, this means Tcl access to the C-based extension API.

Secondly, the script will use the language features such as list and string manipulation and flow-of-control commands.

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

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

And similarly when handling a request completely in C:

<table>
<thead>
<tr>
<th>Component</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>round trip latency</td>
<td>38</td>
</tr>
<tr>
<td>interpreter creation</td>
<td>4</td>
</tr>
<tr>
<td>POST scriptlet</td>
<td>17</td>
</tr>
<tr>
<td>display scriptlet</td>
<td>2</td>
</tr>
<tr>
<td>framework processing</td>
<td>12</td>
</tr>
<tr>
<td>Total response</td>
<td>53</td>
</tr>
</tbody>
</table>

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 in performance-critical situations.

Timing tests were performed on an IXP420-based systems @ 266MHz.
## Timing Comparison

<table>
<thead>
<tr>
<th>Task</th>
<th>C-based</th>
<th>Tcl-Based</th>
</tr>
</thead>
<tbody>
<tr>
<td>round trip latency</td>
<td>38ms</td>
<td>38ms</td>
</tr>
<tr>
<td>interpreter creation</td>
<td>-</td>
<td>4ms</td>
</tr>
<tr>
<td>POST scriptlet</td>
<td>1ms</td>
<td>17ms</td>
</tr>
<tr>
<td>display scriptlet</td>
<td>2ms</td>
<td>2ms</td>
</tr>
<tr>
<td>framework processing</td>
<td>12ms</td>
<td>12ms</td>
</tr>
<tr>
<td><strong>Total response</strong></td>
<td>53ms</td>
<td>73ms</td>
</tr>
</tbody>
</table>
Possible Applications

- control cameras, frame rate, image processing, network access
- environmental sensor data gathering, analysis
- industrial control
Scripting Language Requirements

- 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
• 10x speed of TinyTcl, 50% of Tcl 8.4
• Small (80-150KB)
• Designed for embedding
• BSD licence
Jim Tcl Features

- regular expressions
- exec
- associative arrays, lists
- file, glob, open, close, read, write
- functional programming
- accurate error reporting
- arrays as first class objects
- 64 bit integers
- strings containing nulls
- list expansion operator
- simple packages
- event loop, sockets
Lua

- Designed for embedding
- Portable
- Small
- Byte code
- BSD licence
- Used in World of Warcraft
Other Scripting Languages

- Pawn (formerly Small)
- Pike
- Nesla
Leveraging Scripting
Ad-hoc scripts

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 []:

Simple Menuing System for Internal Use
Leveraging Scripting Prototyping

- Fills the gap between shell scripts and C
- Small daemons
- Configure systems
- Exec commands
- Parse files
- Reload config on SIGHUP
Leveraging Scripting
Replace Complex Shell Scripts

- String and data structure manipulation
- Invocations of sed/awk/grep are slow
- Shell quoting hell
- No floating point math
- Start-up time may be critical
Leveraging Scripting
“Free” CLI

• Easy to add Command Line Interface
• User Interaction
• Debugging

• Possible mechanisms:
  • Unix domain sockets
  • Special startup mode
Pitfalls

- Excessive stack usage
- Unicode support
- No-MMU support
- Licencing
- IP Leaking
More about Jim Tcl

- Expand operator
- List-dictionary duality
- Source location tracking
- Get it:
  - http://jim.workware.net.au/