CriTcl - Beyond Stubs and Compilers

Steve Landers
Digital Smarties
steve@digital-smarties.com

Jean-Claude Wippler
Equi4 Software
jcw@equi4.com

1 Introduction

Scripting can only get you so far. Often the last 5-10% of an application needs to be compiled, for a number of reasons, including

- performance
- to hide proprietary features (such as licensing)
- custom features that can't be implemented in Tcl
- the use of compiled extensions

Fortunately, it is relatively easy to add C extensions to Tcl, either by building custom interpreters or dynamically loadable extensions. But it is still a complex issue - not only does one have to deal with tools such as autoconf, make, etc. - but it also means having a development environment configured with Tcl/Tk header files and "stub libraries". The conceptual difference between "writing a bit of C code" and having an extension which is actually callable from a Tcl script can be daunting. Worse, the Tcl build environment is somewhat foreign to those building on a Windows platform (TEA[1] is not everyone's cup of tea). Yes, "real programmers" will point out that this is what it takes to do "true" programming. But for many who are used to the productivity of Tcl it can easily become a frustrating and time-wasting exercise.

This paper describes CriTcl - which stands for "**C**ompiled **R**untime **I**n **Tcl**" - a tool designed to address these issues. CriTcl allows C code to be embedded in Tcl scripts, (transparently) compiled and called like any Tcl procedure.

CriTcl also takes care of many of the details of building compiled extensions, hiding issues such as including Tcl headers, stubs support and creating Tcl commands for C functions - even setting up fully TIP 55[2] compliant packages in a directory, with a cross-platform pkgIndex.tcl ready to go.

2 A Brief History of CriTcl

CriTcl started life as an experiment by Jean-Claude Wippler and was a self-contained Tcl package to build C code into a Tcl/Tk extension on the fly. It was somewhat inspired by Brian Ingerson's Inline for Perl[3], but is considerably more lightweight.

[1] Tcl Extension Architecture [1]

[2] TIP #55 Package format for Tcl extensions - specifies the contents of a binary distribution of a Tcl package, suitable for automated installation into an existing Tcl installation [2]

The original idea was to wrap C code into something that could then be compiled into a Tcl extension, (if necessary) compile it using gcc when the end user runs the application and then dynamically load it into the running Tcl/Tk interpreter. CriTcl used a MD5 checksum of the source code to decide if it needed recompiling, so subsequent use was almost instantaneous. Both the MD5 checksum for the generated source and the compiled shared library are cached.

The use of Tcl stubs, and the fact that CriTcl had all the include files it needs to make compilation self-contained, means that this was a pure Tcl package, which worked with any (8.1 and up) installation of Tcl. Most importantly, CriTcl did not care a bit where Tcl was installed, nor even whether it was built as a static or as a dynamic executable.

So, in one sense, CriTcl (the library) permitted Tcl scripting to go that last few percent where raw performance becomes an issue.

The second chapter of the CriTcl story is CritLib[4] - a set of Tcl extensions - and CritBind - a utility script that used the CriTcl package to build a shared library out of the CriTcl extensions.

This takes the CriTcl story another step further - now, the developer could use CriTcl to build common libraries, which are then available for deployment in environments without a C compiler. CriTcl still required gcc - but at least this is commonly available to developers. And it is potentially only a small step to any C compiler with a command line interface.

CritLib provided valuable experience in building common compiled Tcl/Tk extensions. This includes blowfish, mathf, md5c, TkHtml and zlib. But CritBind was still relatively difficult to use, and so the third chapter of the CriTcl story began.

CriTcl (the command) makes CriTcl trivially easy to use. For convenience, it is deployed as a Starkit [5] (although it can be unwrapped and used with a more traditional Tcl interpreter). It can be run in one of three modes - a "compile and run" mode (similar to the original CriTcl experiment); a "build library" mode (similar to CritBind); and a "package" mode.

It is this latest incarnation of CriTcl that will be discussed in detail by this paper - with particular focus on the package mode for generation of ready-to-run Tcl extensions for multiple platforms.

3 CriTcl Overview

As mentioned, CriTcl allows C code to be embedded in Tcl scripts, (transparently) compiled and called like any Tcl procedure.

It provides procedures for:

- defining C functions that are callable as Tcl procedures
- specifying header files and C command line parameters
- injecting C code into the intermediate C program
- checking if CriTcl is available on the current platform
- forcing a compile and checking whether it was successful

All generated code and the MD5 checksums are collected in a per-user/per-platform directory (~/.critcl/*platform*) - which can be safely located on a shared volume.

CriTcl is self-contained:

- there are copies of all required Tcl and Tk header files inside CriTcl, so there is no need for a developer Tcl/Tk installation
- there is no need for pre-compiled Tcl/Tk stub libraries to link against, since CriTcl has its own stub glue included

For convenience, CriTcl is packaged as a Starkit - a single file packaging of Tcl code and data - which can be downloaded from the Starkit Distribution Archive [6] .

The following examples will demonstrate the use of CriTcl in a number of situations.

3.1 Extending Tcl Scripts with C code

Consider the following Tcl script, which performs some simple calculations and prints the time taken to perform each one:

```
proc sum {a b} {
     return [expr {$a+$b}]
}
proc pow3 {a} {
     return [expr {$a*$a*$a}]
}

proc timeit {txt cmd} {
  set num 100
  set count 1000
  set run {}
  for {set n 0} {$n < 100} {incr n} {
      lappend run $cmd
  }
  set val [uplevel 1 [list time \
            [join $run {; }] $count]]
  set tmp [lreplace $val 0 0 \
     [expr {[lindex $val 0]/(1.0*$num)}]]
  puts "$txt: [lrange $tmp 0 1]"
}

set a 1 ; set b 2
timeit "Tcl noop" {}
timeit "Tcl expr" {expr {1+2}}
timeit "Tcl vars" {expr {$a+$b}}
timeit "Tcl sum " {sum 1 2}
timeit "Tcl expr" {expr {2*2*2}}
timeit "Tcl vars" {expr {$b*$b*$b}}
timeit "Tcl pow3" {pow3 2}
```

Note the **sum** and **pow3** procedures that we will be timing. The **timeit** procedure runs each test 100 times, invoking the interpreter 1000 times to get an average value per iteration.

If we save the above script in a file **time.tcl** and run it using a Tcl 8.4 interpreter on an Apple Mac iBook 500Mhz, it gives the following:

```
$ tclkit time.tcl
Tcl noop: 0.01 microseconds
Tcl expr: 0.75 microseconds
Tcl vars: 1.8 microseconds
Tcl sum : 4.46 microseconds
Tcl expr: 1.08 microseconds
Tcl vars: 2.8 microseconds
Tcl pow3: 4.98 microseconds
```

Now, we can start adding some C code to the top of **time.tcl** using CriTcl. We'll define three C functions:

- **noop** - which demonstrates why scripted no-ops are quicker than compiled ones :□)
- **add** - equivalent to the **sum** Tcl procedure
- **cube** - equivalent to the **pow3** Tcl procedure

```
package require critcl

critcl::cproc noop {} void {}

critcl::cproc add {int x int y} int {
  return x + y;
}

critcl::cproc cube {int x} int {
  return x * x * x;
}
```

Note that the **critcl::cproc** procedure defines a C function, and sets up the corresponding Tcl command. We can now add the commands to invoke these to end of the **time.tcl** script:

```
timeit "  C noop" noop
timeit "  C sum " {add 1 2}
timeit "  C vars" {add $a $b}
timeit "  C pow3" {cube 2}
timeit "  C vars" {cube $b}
```

Running the new version of **time.tcl** using the CriTcl Starkit gives the following results:

```
$ critcl time.tcl
Tcl noop: 0.02 microseconds
Tcl expr: 0.78 microseconds
Tcl vars: 1.86 microseconds
Tcl sum : 4.49 microseconds
Tcl expr: 0.97 microseconds
Tcl vars: 2.63 microseconds
Tcl pow3: 4.58 microseconds
  C noop: 2.96 microseconds
  C sum : 2.07 microseconds
  C vars: 3.22 microseconds
  C pow3: 1.94 microseconds
  C vars: 2.52 microseconds
```

When running this the first time, there is a pause for a few seconds after the last of the Tcl test results is printed, while the generated code for the C functions is compiled and the corresponding shared library is cached. On subsequent runs there is no discernible pause because CriTcl detects that the source hasn't changed and just loads the shared library.

This example demonstrates the ease with which C code can be added to a Tcl script, and shows the corresponding speed up.

3.2 Building Libraries

CriTcl can also be used to generate a dynamically loadable shared library, so the code can be run on machines without a C compiler present.

If we wanted to build a small shared library containing just the above math functions, we could create a file **mymath.tcl** containing the CriTcl declarations, and then build it using:

```
$ critcl -lib mymath.tcl
Source: mymath.tcl
Library: mymath.dylib
```

We are left with a mymath shared library (in this case a MacOS X .dylib, but on Windows a .dll and on most other Unices a .so) which we can invoke using:

```
$ tclkit
% load mymath.[info sharedlibextension]
% add 1 2
3
% cube 2
8
```

3.3 Building Packages

But shared libraries are much more convenient when distributed using the Tcl package facility. Using this mode, CriTcl will generate a Tcl package structure (including the pkgIndex.tcl file). The structure is arranged so that the results of invocations on different architectures are accumulated in platform specific directories and when run, the pkgIndex.tcl file loads the shared library for the current platform. These package structures are useful for creating cross-platform applications - such as Starkits that may be run on several architectures.

So, let's update our **mymath.tcl** with a "package provide" command:

```
package require critcl
package provide mymath 1.0

critcl::cproc noop {} void {}

critcl::cproc add {int x int y} int {
  return x + y;
}

critcl::cproc cube {int x} int {
  return x * x * x;
}
```

and build the package using:

```
$ critcl -pkg mymath
Source: mymath.tcl
Library: mymath.dylib
Package: lib/mymath
```

(Note that the **.tcl** extension is optional on the CriTcl command line).

Copying the **lib/mymath** directory to somewhere on Tcl's auto_path, or to the **lib** directory of a Starkit, will allow the **mymath** package to be loaded as follows:

```
$ tclkit
% package require mymath
1.0
% add 1 2
3
```

As we have seen, CriTcl generates a **lib/mymath** directory next to the **mymath.tcl** source. Looking inside this we see:

```
lib
`-- mymath
    |-- critcl.tcl
    |-- pkgIndex.tcl
    `-- Darwin-ppc
        |-- critcl.tcl
        `-- mymath.dylib
```

Firstly, note that there are two Tcl scripts - critcl.tcl and pkgIndex.tcl. The first contains scripts used by the latter:

```
source [file join $dir critcl.tcl]
critcl::loadlib $dir. mymath 1.0
```

The critcl.tcl file is more complex - it provides dummy definitions for each CriTcl procedure (in case they are called from the other scripts) plus two key procedures:

- **critcl::loadlib** which is used by pkgIndex.tcl to load the appropriate shared library and any Tcl files
- **critcl::platform** which generates a normalised name for each platform (needed because the contents of tcl_platform array are not always consistent across different architectures)

Note also that there is a platform specific directory containing the shared library, plus a small critcl.tcl script that stores the values of certain CriTcl parameters from the build process, in case they are referred to by the package Tcl script.

As the package is built on multiple platforms, CriTcl will add similar directories. For example, if we copy mymath.tcl and the lib directory to a Linux platform and run CriTcl again:

```
$ critcl -pkg mymath
Source: mymath.tcl
Library: mymath.so
Package: lib/mymath
```

CriTcl takes care to store the cached and intermediate files in separate per-user/per-platform directories, so it would be safe to have both the **mymath** source and the user's home directory on a shared network drive.

3.4 A More Complex Package

Writing packages using a self-contained Tcl script is one thing, but what about more complex packages with external source files?

As an example, we'll look at making Eric Young's implementation of Blowfish encryption library [7] available as a Tcl package.

Firstly, we put the C source and headers for blowfish into a separate directory **blowfish_c**:

```
$ ls blowfish_c
bf_cfb64.c bf_enc.c  bf_locl.h
bf_pi.h    bf_skey.c blowfish.h
```

Note that these files are unchanged from the original distribution.

Now, we create the **blowfish.tcl**[3] CriTcl script to create Tcl commands that invoke the Blowfish library functionality. We'll also start to use some of the more advanced CriTcl facilities.

Firstly, we need to get the CriTcl package, but we'll also check to see that compiling is supported by CriTcl on the current platform:

```
package require critcl
if {![critcl::compiling]} {
    puts stderr "This package cannot \
                 be compiled without  \
                 critcl enabled"
    exit 1
}

package provide blowfish 0.10
```

The **critcl::compiling** procedure returns 1 if CriTcl is able to find a suitable compiler on the current system. There is a companion procedure **critcl::scripting** which returns the inverse of critcl::compiling. Why? Depending on what you are doing it helps make some scripts (maybe even the one above) more readable.

Now we declare the C sources and headers needed when compiling blowfish:

```
namespace eval blowfish {

    critcl::cheaders blowfish_c/*.h
    critcl::csources blowfish_c/*.c

    critcl::ccode {
       #include "blowfish.h"
    }
```

critcl::cheaders lets CriTcl know that the specified files will be required when compiling the package, and will need to be copied to the CriTcl cache.

critcl::csources specifies files that need to be compiled by CriTcl when the package is build.

[3] Note that the full text of blowfish.tcl is shown in Appendix 1

critcl::ccode is used to inject the specified C code into the generated C source file. In this case, the code ensures that the blowfish library declarations are included before compiling any C functions.

Finally we create a C function to implement the blowfish Tcl command (this has been abbreviated - for the full listing see Appendix 1). We do this using the **critcl::ccommand** procedure, which is similar to **critcl::cproc** but at a lower level - it ties C code to an objectified Tcl command (via the Tcl_CreateObjCommand function) without any further wrapping:

```
namespace blowfish {
    ...
    critcl::ccommand blowfish \
                    {dummy ip objc objv} {
        int index, dlen, klen, count = 0;
        unsigned char *data, *key;
        Tcl_Obj *obj;
        ...
        Tcl_SetObjResult(ip, obj);
        return TCL_OK;
    }
}
```

And that's it! We just need to compile and test - this time on Linux:

```
$ critcl -pkg. blowfish
Source: blowfish.tcl
Library: blowfish.so
Package: lib/blowfish
$ tclkit
% lappend auto_path [pwd]/lib
/usr/local/bin/tclkit/lib/tcl8.4 \
/usr/local/bin/tclkit/lib \
/home/steve/blowfish/lib
% package require blowfish
0.10
% set plain "Hello world!"
% set key "this is a secret"
% set coded [blowfish::blowfish \
                    encode $plain $key]
% binary scan $coded H* secret
% puts "coded = $secret"
coded = 258d3a52bf61df2467bade73
% puts "clear = [blowfish::blowfish \
                    decode $coded $key]"
clear = Hello world!
```

We can build on additional platforms, accumulating the contents of the **lib** directory as we go. This will give us a **lib** directory like the following:

```
lib
`-- blowfish
    |-- critcl.tcl
    |-- pkgIndex.tcl
    |-- Darwin-ppc
    |   |-- blowfish.dylib
    |   `-- critcl.tcl
    |-- Linux-x86
    |   |-- blowfish.so
    |   `-- critcl.tcl
    `-- Solaris-sparc
        |-- blowfish.so
        `-- critcl.tcl
```

Note that there are other tools that allow you to build interfaces between Tcl and external libraries, in particular, the SWIG - Simplified Wrapper and Interface Generator [8].

SWIG provides a greater level of automation than CriTcl. When using SWIG the developer prepares a small interface file that specifies what functions are to be wrapped. This is used by SWIG to generate the "glue" between an external library and Tcl - the equivalent of the **critcl::ccommand** is generated for you. On the other hand CriTcl is significantly easier to deploy, and generates a package structure ready to use.

3.5 A Tk Package

Building a Tk extension using CriTcl is slightly more complex.

Consider the **Tkspline** package by John Ellson - part of the Graphviz software [9] . Tkspline provides an additional line smoothing method for the Tk canvas widget, and is typically used in conjunction with **Tcldot** - the graphviz package for drawing directed graphs on a canvas . As distributed, Tkspline is built as part of the Graphviz build system, which is large and complex. By extracting Tkspline and building it using CriTcl it becomes feasible to move it to platforms not supported by Graphviz (Windows, for example) and makes support much easier.

So, we'll start constructing **Tkspline.tcl** (the capitalisation was necessary to force the CriTcl generated package to have the same name as the original Tkspline package). Here some of the details will be omitted, but the full text of the script is shown in Appendix 2:

```
package require critcl
package provide Tkspline 0.4.1

critcl::tk

set tcl_prefix [file normalize ~/src/tcl]
set tk_prefix  [file normalize ~/src/tk]

critcl::cheaders \
     -I$tk_prefix/generic -I$tk_prefix \
     -I$tcl_prefix/generic -I$tcl_prefix
```

The **critcl::tk** procedure tells CriTcl to include the Tk stubs table setup in any generated code, plus add **tk.h** to the list of C headers files.

But Tkspline also requires some of the internal Tcl and Tk header files (**tclInt.h and tkInt.h) -** so the next lines specify the location of the Tcl and Tk source trees (it would be preferably for CriTcl to have a means for these to be specified on the command line, or even default to a location like /usr/local/src/tcl):

```
set tcl_prefix [file normalize ~/src/tcl]
set tk_prefix  [file normalize ~/src/tk]

critcl::cheaders \
     -I$tk_prefix/generic -I$tk_prefix \
     -I$tcl_prefix/generic -I$tcl_prefix
```

The last three lines add the specified Tcl and Tk directories to the list of directories searched for included header files (note that this is different from the use of **critcl::cheaders** shown in the earlier examples). Whilst the use of internal Tcl/Tk header files makes the building of Tkspline potentially Tcl/Tk version specific - it is nevertheless much simpler to deal with than the autoconf, makefiles and other complexities of the standard Graphviz build environment.

Also note also that we are now starting to include build information in the CriTcl script (more on the build features later). But what about handling platform differences? Fortunately, this is one of the (many) things that Tcl does well ...

```
#
#    platform specific declarations
#
switch $tcl_platform(platform) {
  unix {
     switch $tcl_platform(os) {
        Darwin  {
           set xinclude /usr/X11R6/include
           set xlib /usr/X11R6/lib
        }
        default {
           set xinclude /usr/X11R6/lib
           set xlib $xinclude
        }
      }
       critcl::cheaders -I$xinclude
       critcl::clibraries -L$xlib -lX11
     }
     windows {
         critcl::cheaders -DWIN32 \
              -I$tk_prefix/win \
              -I$tk_prefix/xlib
     }
     default {
          puts stderr "tkspline hasn't been \
               ported to \
             $tcl_platform(platform)"
          exit 1
     }
}
```

The only new CriTcl procedure is the **critcl::clibraries** - this adds the specified commands to the C compiler command line (in this case, ensuring that the X11 runtime library is resolved when building the package shared library).

And finally, we include the original Tkspline C code, arranging for the original Tkspline_Init function to be called:

```
critcl::ccode {
     #define Tkspline_Init ns_Tkspline_Init
     #include "tkspline.c"
}

critcl::cinit {
     Tk_CreateSmoothMethod(ip,
               &splineSmoothMethod);
}
```

The first line in the **critcl::ccode** invocation is a small trick to make the Tkspline C code expect the same shared library initialisation routine as that generated by CriTcl. After

that, we just include the Tkspline code unchanged. The **critcl::cinit** procedure injects code into the shared library initialisation function - in this case we want to add Tkspline as a canvas smoothing method.

So now, all that is left is compiling in the usual way:

```
$ critcl -pkg. Tkspline.tcl
Source: Tkspline.tcl
Library: Tkspline.so
Package: lib/Tkspline
```

4 Advanced Features

The previous example showed how we can use standard Tcl features along with CriTcl to add simple build information to a script. But CriTcl allows you to go further - for example, testing for compiler or platform features, or checking for a failed compile.

4.1 Checking C Code

With more complicated extensions - especially Tk extensions - it is often necessary to check for platform specific features.

For example, the busy widget has been extracted from George Howlett's BLT library [10] and built as a separate stubs-enabled package using CriTcl. Here is a code fragment from busy.tcl:

```
if {[critcl::check "#include <limits.h>"]}{
     critcl::cheaders -DHAVE_LIMITS_H
}
```

The **critcl::check** procedure invokes the C compiler on the supplied C code, and returns true if the code compiles without errors. In this case, we are just testing for the existence of the **limits.h** header file.

Using the C compiler in this way is perhaps less efficient that the more traditional approaches (such as autoconf). But it does have some key advantages - primarily simplicity and familiarity. Knowing C does not imply a knowledge of any particular compiler tool chain, especially as more people come to Tcl from the Windows and Mac worlds.

4.2 Fallback to Tcl

Now that using C is less daunting for the average Tcl programmer, "plug-in replacement by C code" becomes a valid strategy. But sometimes we can't use a compiled version on a given platform - e.g. if there is no compiler available on the platform, or if the C code uses features that are not portable. In this case it is important to be able to fall back to a pure-Tcl implementation - even if it is slower, or perhaps missing some features. The aim is to always end up with some working code, even if that code is nothing more than a clearly expressed failure explanation.

To see how this is done, let's look again at the first example, and extended the **mymath** example so that it will fall back to a Tcl implementation if a C compiler isn't available, or if the compile fails for any reason:

```
package provide mymath 1.0
package require critcl

proc fallback {} {
     proc ::noop {} {}
     proc ::add {x y} {
          return [expr {$x + $y}]
     }
     proc ::cube {x} {
          return [expr {$x*$x*$x}]
}
if {[critcl::scripting]} {
     fallback
} else {
     critcl::cproc noop {} void {}
     critcl::cproc add {int x int y} int {
        return x + y;
     }
     critcl::cproc cube {int x} int {
          return x * x * x;
     }
     if {[critcl::failed]} {
          fallback
     }
}
```

The **critcl::failed** procedure can be called once within a CriTcl script. It forces a compilation of the generated C code and returns true if the compilation (or link) fails. This is typically used at the end of a CriTcl script to either fall back to Tcl code and/or to issue a warning message. Note that invoking critcl::failed stops any compiler errors being displayed (they can still be viewed in the CriTcl log file under ~/.critcl/*platform*.

This fallback approach could be useful in projects such as Tcllib[11], which must be guaranteed to work on platforms without a C compiler but that contain modules which could benefit from the speed of a C implementation.

4.3 Dual Tcl and C Implementations

When building packages using CriTcl, it is still desirable to implement as much as possible in Tcl. To help achieve this, CriTcl provides the **critcl::tsources** command to list Tcl scripts that need to be included in the generated package.

For example:

```
critcl::tsources src1.tcl src2.tcl
```

When building the package, CriTcl will copy these scripts to a separate directory under package directory (e.g. lib/*package*/Tcl) and CriTcl will source these files when the package is loaded.

4.4 Cross Compiling

As well as supporting compiling on Windows via Cygwin [12] or Mingw[13], CriTcl supports cross compiling libraries and packages for Windows on Linux/Unix using the Xmingwin cross compiler (which is based on Mingw) . When CriTcl starts it first tries to recognise a cross-compile environment by looking at the version of the C compiler being used (by running "gcc -v"). It should be relatively

straightforward to extend this to any gcc based cross compiler, allowing CriTcl to build code on (and for) most major platforms.

To set up an Xmingwin cross compiler use the script posted to comp.lang.tcl by Mo de Jong (a copy is available at [14]). This script (based on an original version by Mumit Khan) downloads Xmingwin, builds it and installs it locally under /usr/local/Xmingwin (you can edit the script and change the PREFIX variable if you want to install it elsewhere). Once you have Xmingwin installed, you'll need to set your PATH to include the Xmingwin bin directory before using CriTcl. One convenient way of doing this is to create a script called *cross* (in /usr/local/bin or ~/bin):

```
$ cat /usr/local/bin/cross
PATH=/usr/local/Xmingwin/i386mingw32msvc/bi
n:$PATH
export PATH
exec $@
```

Then, you can compile using the usual CriTcl package (or library) building commands:

```
$ cross critcl -pkg blowfish
Cross compiling for Windows using Xmingwin
Source: blowfish.tcl
Library: blowfish.dll
Package: lib/blowfish
```

If CriTcl recognises a cross compile environment, it manipulates the tcl_platform array so that it matches that found on Windows 2000. Specifically, the following values are set:

```
tcl_platform(byteOrder)  = littleEndian
tcl_platform(machine)    = intel
tcl_platform(os)         = Windows NT
tcl_platform(osVersion)  = 5.0
tcl_platform(platform)   = windows
tcl_platform(wordSize)   = 4
```

Critcl also provides the **critcl::sharedlibext** procedure, which returns the shared library extension for the target platform. If you plan on cross-compiling you should use this variable in your CriTcl scripts instead of **info sharedlibextension** (although overlaying the **info sharedlibextension** command will probably happen at some stage).

Intermediate files are stored in **~/.critcl/Windows-x86** irrespective of the platform on which cross compiling occurs.

There are two other CriTcl procedures that are useful in this context - **critcl::platform** returns the target platform, and **critcl::cache** returns the name of the directory where CriTcl intermediate files are stored.

4.5 Lower Level Stuff

Critcl::cproc arguments and return values must be typed. There are no default arguments or ways to pass more sophisticated data items than int/long/float/double /char*. The return type can be “string”, meaning that it is a Tcl_Alloc’ed char* which will be Tcl_Free’d at some point.

You can, however, use Tcl_Obj* arguments (with no refcount change), or return it (in which case it will be decref'ed). If the first parameter to **critcl::cproc** has type **Tcl_Interp*** that will be passed in. Lastly, if the return type is **ok**, then an integer return code of type **TCL_OK** or **TCL_ERROR** is expected and will be processed as usual (errors must set the result, so it is most likely that you'll also want to specify the **Tcl_Interp*** argument).
The **critcl::cdata** procedure can be used to create a Tcl byte array object, and associated Tcl command:

```
critcl::cdata hi “hi there!”

% hi
hi there!
```

When using **critcl::ccommand** (for example, to invoke existing C functions in a library) it is possible to specify the **clientdata** and **delproc** arguments to the generated Tcl_CreateObjCommand function. To do this, use the optional **-clientdata** and/or **-delproc** before the argument specifying the C code.

For example, the following is from the CriTcl wrapping of the BLT busy widget:

```
critcl::ccommand release \
             {data ip objc objv} \
             -clientdata BusyDataPtr {
    int argc = objc;
    char **argv = obj_convert(objc, objv);
    ThreadData *dataPtr =  \
           (ThreadData *) data;
    Busy *busyPtr;
    int i;

    for (i = 1; i < argc; i++) {
        if (GetBusy(dataPtr, ip, argv[i],\
                  &busyPtr) != TCL_OK) {
            return TCL_ERROR;
        }
        HideBusyWindow(busyPtr);
            busyPtr->isBusy = FALSE;
        }
    Tcl_SetResult(ip, NULL, NULL);
    return TCL_OK;
}
```

4.6 Solving Compile Problems

The intermediate C code generated by CriTcl can be kept around by using the **-keep** flag. It is also possible to force a compile (i.e. disregarding the MD5 checksum) by using the **-force** flag:

```
$ critcl -force -keep -pkg blowfish
```

In case of compile errors the source always remains in the ~/.critcl/*platform* directory, but either way it won't be obvious because of the file names generated from the MD5 hash. To find out what the last compile was doing, look at

the end of the log file, which will have a name like v032.log

CriTcl inserts a "#line" directive in the generated C source, so that an error on line three of **cproc** "foo" in script "bar.tcl" will be reported by gcc as occurring on "bar.tcl/foo", line 3. No name is added to **ccode** sections, so with multiple sections, identifying the line is harder.

Critcl::cheaders can be used to set compile or linker flags, for example:

```
$ critcl::cheaders -g
```

causes the output library to contain symbols and the **-DNDEBUG** flag is not added to the gcc command line.

One area that will need to be addressed is the debugging of CriTcl procedures - even if it is just defining a few procedures to do "puts-style" debugging in C.

4.7 Cleaning Up

Over time, the ~/.critcl/ directory could fill up with debris - log files, old builds of compiled code that failed, and libraries for various platforms.

You can always delete the ~/.critcl/ area - it has no further impact than causing a few recompiles on next use.

5 Work In Progress

CriTcl is relatively new technology and there are an increasing number of people testing the limits and taking CriTcl forward in a number of areas.

5.1 Additional Languages and Compilers

Arjen Markus has started work on modularising the compiler interface - to allow definition of compilers, linkers and other build tools in a generic way. This has been used to provide a “proof-of-concept” version of CriTcl that works on Windows with Microsoft Visual C. It could equally be used to interface with proprietary C compilers, which often deliver better object code that gcc (for example, the Solaris C compiler).

But it turns out that CriTcl is not just a way to bind to C code, it just offers a means to wrap compilation of other languages, with a bit of automation for all glue aspects. Arjen Markus has also used the modular build system to in apply the concepts of CriTcl binding with Fortran source code:

```
critclf::fproc nop {} void {
    return
}

critclf::fproc iadd {int x int y} int {
    iadd = x + y
    return
}
critclf::fproc cube {int x} int {
    cube = x * x * x
    return
}
```

Increasingly, however, users do not have a C compiler available nor the time or inclination to install one. This is especially true of Windows users. Work is currently underway to build a suite of gcc cross-compilers for common platforms (most likely Windows, Linux and MacOS X). These would be self-contained, stripped of unnecessary components to reduce the size and installed in a standard location that CriTcl knows about. This would make it much easier for a novice CriTcl user **on** any of the above platforms to generate packages **for** any of the platforms.

Alternatively, it may be possible to include a small compiler inside CriTcl - for the x86 architecture - so that for Windows and Linux no separate compiler is needed. This would be of benefit even if the included compiler generates inferior code, i.e. one could choose between "easy installation" and "more sophisticated compiler".

Another alternative being considered is to specify that a Tcl procedure should be compiled to bytecodes, and the bytecode included in the generated package. The incentive is not performance, but privacy. This would transparently integrate TclPro’s [15] procomp and tbcload facilities.

For example,

```
critclf::tcomp myproc {a1 a2} {
    # this will become bytecodes
    for {set i $a1} {$i < $a2} {incr i} {
        # do something sneaky
        ...
    }
}
```

This would give the best of both worlds - important code could be protected (including, for example, licensing code) but the developer could still use Tcl rather than a compiled language.

5.2 Remote Compilation

It is possible to have CriTcl automatically package the generated C code and all necessary source files as a self-contained archive suitable for copying to a target machine and compiling there.

This could be used as the basis of a "remote compilation" facility, where this packaged source is shipped to another machine running a special "server-mode critcl", which performs the compile and ships back the compiled code. There is an older experiment which demonstrates this approach is at [16].

5.3 Building TclKit

Another area being investigated is streamlining of the build of TclKit itself so it fully relies on CriTcl. This would

make it yet simpler to generate a TclKit executable, using nothing but totally standard distributions, generic code, and some Tcl/CriTcl scripts. A first trial was done end of 2001, and the results were promising.

6 One intriguing possibility ...

CriTcl need not be limited just to Tcl applications and extensions. One idea is to experiment with the concept of "systems scripting". Just as a generation ago systems programming moved from assembler to C, the time may be right to shift to a predominantly scripted model.

The Squeak language [17] has been implemented this way, using C as "assembly" but coding even its own Virtual Machine as Squeak/Smalltalk code which then generates C. And Squeak is no performance slouch - partly because low-level designs can be radically altered with little effort (which the authors did when switching to Forth's Threaded Code model).

Using this approach for Tcl, the first step would be to generate tcl.h (and the other header files) from the stub table definitions themselves, and to generate stub entries (i.e. not just the C functions) for binding to Tcl using the Tcl_CreateObjCommand.

This would make it possible to gradually replace parts of Tcl itself with CriTcl-built code. Taking this last idea further - the Tcl API could be split into "primary" and "secondary" calls, and CriTcl used to build the secondary ones. This would be a first step towards modularisation, and a smaller Tcl core. One could for example take the regex subsystem, turn it into a CriTcl module and make it optional - or at least allow a developer to switch between minimal and full-scale versions (the "xre" example in CritLib shows that regex can indeed be built separately).

More practical, perhaps, would be to use this approach to turn the native file system interface into a module. Other candidates might be the bytecode compiler, the channel subsystem, or the exec/pipe-open/load interfaces. Taken to extremes, the "tcl.h" plus "tcl.decls" headers could become the center of Tcl, with everything else plugged in - leading to an equivalent but deeply modular core, and a workbench for future yet more radical Tcl changes.

It will indeed take some pretty capable coders to implement this approach, but only at the "bottom end". Once the mechanism is in place and a foundation working, everything else can be written in Tcl (albeit a limited subset) by people who know little about C. Then, at some point a Tcl+C expert can go through what has been implemented, pick a hot spot, and replace Tcl by a Tcl/C mix using CriTcl.

Maybe Tcl is not the optimal language for this approach but, on the other hand, maybe it is so malleable that it might just work. Even the OS and compilers themselves would be candidates for this approach - end up with more general and flexible "skeletons" on which everything else rests.

A first step in this direction has been taken recently by Andreas Kupries - the Scripted Compiler project [18]. Using this, one could write a "C to machine code" translator in Tcl and end up with a system which can generate and maintain itself with no compiler at all (except for the usual bootstrap issues).

The trend and motivation in all of the above is not to redo things just for the fun of it, but to attempt to change the core/systems development from a compiled to a scripting model and in doing so, realise the same benefits of productivity, flexibility and extensibility.

7 Conclusion

Tcl started its life as an extension language for C. But in many ways it has become a victim of its own success - for a growing number of scripters C has become an afterthought and inconvenience. Even those who (reluctantly) code in C know lots about Tcl, and very little about compilers, linkers, make, autoconf, etc.

The key benefit of CriTcl is that it reduces barriers to adding C code to application, barriers caused by needing to know too much about Tcl extension structures, build environments, compiler tool chains, etc. And in doing so, it addresses the (often heard but equally often fallacious) concern that "scripting programs are too slow". Rather than build in a compiled language, CriTcl allows an application to be developed in Tcl/Tk - with all the associated productivity and flexibility benefits. In the event that a performance bottleneck is identified, CriTcl allows that bottleneck to be recast in C. But more importantly, it allows this decision to be deferred - there's no sense in scratching before it itches. And by allowing the developer to continue working with Tcl, it increases the chances that performance will addressed at the architectural level.

But CriTcl also takes the Tcl concept of "**glueware**" to new levels. One can start with a Tcl/Tk interpreter (either a traditional tclsh/wish, ActiveTcl or TclKit) and develop with Tcl being at the center and everything else being small pieces of C code. CriTcl becomes the means to "run that last mile", to get performance and to connect to other C code. And it potentially saves a significant amount of time, since there is no need to know about build environments, tool chains, etc.

So, CriTcl has come from being an experiment in defining C procedures for performance, to being a new way of building Tcl extensions, and potentially Tcl itself. And, perhaps, it could become a stepping stone to a leaner, meaner and more modular Tcl/Tk core. Then the circle will have closed - Tcl will no longer be a victim of its own success in removing the need for compiled applications.

Acknowledgement

Thanks are due to Mark Roseman for his patient reading of the draft of this paper, his insightful comments and his uncanny ability to spot even the smallest typo. Thanks also to Larry Virden for generously giving his time.

References

[1] TEA - *The Tcl Extension Architecture* http://www.tcl.tk/doc/tea

[2] Cassidy, Steve *TIP #55 - Package Format for Tcl Extensions* http://www.tcl.tk/cgi-bin/tct/tip/55.html

[3] Ingerson, Brian - *Inline for Perl* http://inline.perl.org

[4] Wippler, Jean-Claude - *CritLib* http://www.equi4.com/critlib

[5] Starkit - http://equi4.com/starkit

[6] The Starkit Distribution Archive - http://mini.net/sdarchive

[7] Blowfish - part of the SSLeay package - ftp://ftp.psy.uq.oz.au/pub/Crypto/SSL

[8] Beazley, David M. - *Tcl and SWIG as a C/C++ Development Tool* - http://www.swig.org/papers/Tcl98/TclChap.html

[9] The Graphviz Graphics Visualization Tookit - http://www.graphviz.org

[10] The BLT Toolkit - http://sourceforge.net/projects/blt/

[11] Tcllib - The Standard Tcl Library - http://www.tcl.tk/software/tcllib

[12] Cygwin - http://www.cygwin.com

[13] Mingw - Minimalist GNU for Windows - http://www.mingw.org

[14] Xmingwin setup script - http://mini.net/sdarchive/xmingwin.sh

[15] TclPro - http://tclpro.sf.net

[16] Wobble - http://wiki.tcl.tk/wobble

[17] The Squeak Programming System - http://www.squeak.org

[18] Scripted compiler - http://wiki.tcl.tk/3687

Appendix 1 - blowfish.tcl script

```
package provide blowfish 0.10
package require critcl

if {![critcl::compiling]} {
    puts stderr "This package cannot be \
          compiled without critcl enabled"
    exit 1
}

namespace eval blowfish {

    critcl::cheaders blowfish_c/*.h
    critcl::csources blowfish_c/*.c

    critcl::ccode {
      #include "blowfish.h"
    }

    critcl::ccommand blowfish \
                     {dummy ip objc objv} {
      int index, dlen, klen, count = 0;
      unsigned char *data, *key;
      Tcl_Obj *obj;
      BF_KEY kbuf;
      unsigned char ivec[] =
   {0xfe,0xdc,0xba,0x98,0x76,0x54,0x32,0x10};
      static char* cmds[] = { "encode",
                     "decode", NULL };
      if (objc != 4) {
        Tcl_WrongNumArgs(ip, 1, objv,
                     "mode data key");
        return TCL_ERROR;
      }

      if (Tcl_GetIndexFromObj(ip, objv[1], \
        cmds, "option", 0, &index) != TCL_OK)
        return TCL_ERROR;

      obj = objv[2];
      if (Tcl_IsShared(obj))
        obj = Tcl_DuplicateObj(obj);
        data = Tcl_GetByteArrayFromObj(obj,
                                       &dlen);
      key = Tcl_GetByteArrayFromObj(objv[3],
                                       &klen);

      BF_set_key(&kbuf, klen, key);
      BF_cfb64_encrypt(data, data, dlen,
                     &kbuf, ivec, &count,
       index == 0 ? BF_ENCRYPT : BF_DECRYPT);

      Tcl_SetObjResult(ip, obj);
      return TCL_OK;
    }
}
```

Appendix 2 - Tkspline.tcl script

```
package provide Tkspline 0.4.1
package require critcl

if {![critcl::compiling]} {
    puts stderr "This extension cannot be \
          compiled without critcl enabled"
    exit 1
}

critcl::tk

set tcl_prefix [file normalize ~/src/tcl]
set tk_prefix  [file normalize ~/src/tk]

critcl::cheaders -I$tk_prefix/generic \
                 -I$tk_prefix \
                 -I$tcl_prefix/generic \
                 -I$tcl_prefix

#
#   platform specific declarations
#
switch $tcl_platform(platform) {
    unix {
        switch $tcl_platform(os) {
            Darwin  {
                set xinclude \
                    /usr/X11R6/include
                set xlib /usr/X11R6/lib
            }
            default {
                set xinclude /usr/X11R6/lib
                set xlib $xinclude
            }
        }
        critcl::cheaders -I$xinclude
        critcl::clibraries -L$xlib -lX11
    }
    windows {
        critcl::cheaders -DWIN32 \
              -I$tk_prefix/win \
              -I$tk_prefix/xlib
    }
    default {
        puts stderr "tkspline hasn't been \
           ported to $tcl_platform(platform)"
        exit 1
    }
}

critcl::ccode {
    #define Tkspline_Init ns_Tkspline_Init
    #include "tkspline.c"
}

critcl::cinit {
    Tk_CreateSmoothMethod(ip,
               &splineSmoothMethod);
}
```