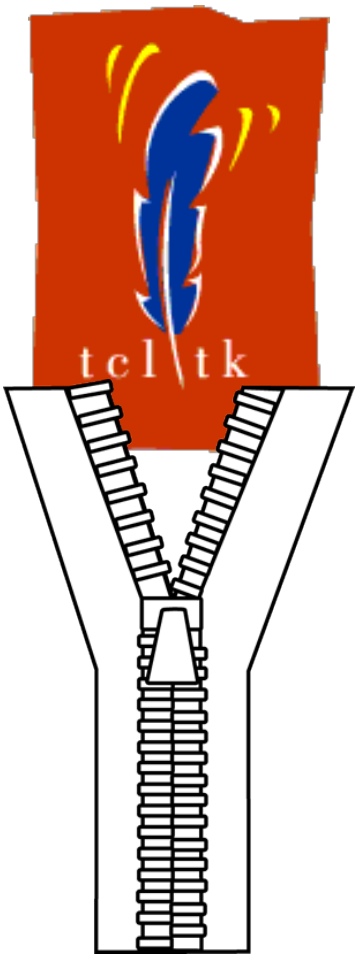

Core ZipVFS

Adding facilities to the Tcl/Tk core to facilitate the building of self-contained executables

Presented at the 21st Annual Tcl Developer's Conference (Tcl'2014)
Portland, Oregon
November 10th-14th, 2014

Sean Deely Woods
Senior Developer
Test and Evaluations Solutions, LLC
400 Holiday Court
Suite 204
Warrenton, VA 20185

Abstract

This paper will discuss leveraging the new Zlib capabilities of the Tcl core with previous efforts (in particular TOBE) to provide a means for attaching Zip files to normal Tcl shells. This work would be of interest to developers of end user software, as well as makers of system utilities.

Problem Statement

Creating self-contained executables in Tcl/Tk is not something that Tcl will do out of the box. While it is possible to statically build a Tcl shell, there is always the problem of **init.tcl**. For reasons great, small and powerful, the startup sequence for an interpreter does a lot of its work in Tcl script. This is normally accomplished by placing init.tcl in the installation path for Tcl/Tk.

For Tcl/Tk 8.6, on a generic Unix system with no special prefix configured, init.tcl is in /usr/local/lib/tcl8.6. And if you don't choose to (or lack the admin privileges) to **make install**, the system will fall back to [path to source]/library.

If init.tcl can't be found, the interpreter will panic, and many commands like "string" and "clock" will not operate properly. As if the situation were not confusing enough, many [Li]l[U]nix distributions have their own installations of Tcl/Tk. And more often than not, bundled with it are a variety of customizations to make Tcl play nicely with the local distribution. Modifying or replacing that init.tcl will break other applications in the OS that rely on Tcl/Tk. Relying on the installed init.tcl file is only helpful if that file is for a compatible version of Tcl. (And the situation is even worse for Tk.)

Single File Executables

The answer I wish to put forth is to provide a mechanism for a shell to lug along its own init.tcl. By using ZipVFS, an executable could do more than that; it can take along an entire file system.

Building Self-Contained Executables

I have been using single file executables for several years. Our product started off using a variant of Richard Hipp's "Tcl as One Big Executable (TOBE)." The reasons why are two-fold.

1. The developer who preceded me on this project was Richard Hipp, himself.
2. TOBE is a concept that has stood the test of time. It is the working principle for FreeWrap, mkTclApp, and ZipKits. (Or are they zip-basekits? Or fotchkits? Or…)

TOBE consists of a statically compiled Tcl/Tk shell with a zip archive tacked onto the back of it. Zip archives are built to understand that they may be attached to an executable. That part was easy. To make it work, we need the Tcl executable to understand that it is attached to the zip archive.

Since the advent of TOBE several nice things have been added to the Tcl Core:

1) Tcl has an internal concept of a Virtual File System. Early editions of TOBE, circa 2000, relied on deep wizardry to intercept file system calls and redirect them to the zipfile.
2) Tcl8.6 includes Zlib.
3) The PreInitScript() shim allows the shell to tweak the variables an interpreter will see on startup
4) The **load** command understands how to load dynamically linked packages from a VFS[1]

TOBE back in 2000 required statically linking Zlib to Tcl, a home-rolled Zip file decoder, and statically linking every scrap of C code you intended to use. There was also the small bit about modifying the Tcl/Tk boot process to orchestrate the mounting of the VFS, tricking Tcl into redirecting file access.

Today, I still compile a statically linked Tcl/Tk shell. But the only outside code I need to include is the Zip file decoder. Extensions (including our proprietary software) are now packed in DLL form into the Zip file system. TOBE does not need to modify the Tcl start process, the shims we need are present already.

The ingredients and scripts to automate the build process are open source, and available online at:

http://fossil.etoyoc.com/fossil/odie

Why ZipVFS?

Why put a Zipfile decoder in the core? Why ZipVFS? Why not Mk4VFS? Or CookFS? Or…

My answer is thus: simplicity. ZipVFS is implemented in 1829 lines of C code. Zip archive encoders are ubiquitous. In environments where those encoders are not as ubiquitous, a pure-tcl version is available in Tcllib. To put the cherry on the top, dis-assembling a ZipVFS self-contained executable can be done right from the "unzip" command. For FreeWrap Dennis R. LaBelle has developed a means to encrypt your VFS.

Why in the Core?

I, myself, am often building Tcl straight from trunk. On the Mac, I also find myself having to evaluate several different versions of the same patch level of Tk at once. Here is one scenario that I have encountered:

- A check in from, say, Friday solves a crash I was having.
- On Tuesday I encounter a new crash.
- But I need to be sure this is a new crash, so I have to test against:
  - A check in (post 8.6.2) that was made on Saturday
  - A check in (post 8.6.2) late Monday night.
  - The real 8.6.2
  - A golden patch of 8.6.1 that contains a hotfix that allows Tk to compile after XCode 5.1 yanked Garbage Collection support.

Comparing multiple versions of the same patch level is relatively simple. It's the complexity of having 8.6.1 and 8.6.2 living on the same machine that causes issues. And it's all down to one line in /usr/local/lib/tk8.6/tk.tcl:

```
package require -exact 8.6.2
```

[1] Or at the very least, copy the DLL out of the VFS and load it from local storage in a manner that is transparent to both the developer and the end user.

Implementation

Checked into the Tcl/Tk fossil tree is a "core_zip_vfs" branch. (One branch for Tcl, one branch for Tk.) Each of those branches contains all of the modifications required to produce a shell that supports ZipVFS. If you follow the development timeline, you will see that I have tried a number of approaches along the way. This paper will summarize what I've done, what worked, and what still needs some work. I will also discuss some alternatives to ZipVFS that may work better for certain applications.

First an overview of the files added:

Filename	Description
doc/zvfs.n	Man page for zvfs
tcl/library/zvfstools/pkgIndex.tcl	Package index for the zvfs utilities
tcl/library/zvfstools/zvfstools.tcl	A pure-tcl zipfile encode and decoder.
tcl/generic/tclZipVfs.c	C implementation of the Zip VFS driver
tcl/generic/tclZipVfsBoot.c	The Shell's boot loader
tcl/tools/mkVfs.tcl	A script to generate the VFS for the Tcl core
tcl/tools/mkzip.tcl	A script to wrap the encoder for zvfstools.tcl

Files modified:

Filename	Description
tcl/doc/tclsh.1	Added notes on zip features
tcl/unix/Makefile.in	Added build recipes for VFS enabled shell
tcl/unix/tclAppInit.c	Added ZipVFS boot loader behaviors
tcl/win/Makefile.in	Added build recipes for VFS enabled shells
tcl/win/tclAppInit.c	Added ZipVFS boot loader behaviors
tk/unix/Makefile.in	Added build recipes for VFS enable shells
tk/unix/tkAppInit.c	Added ZipVFS boot loader behaviors
tk/win/Makefile.in	Added build recipes for VFS enabled shells
tk/win/winMain.c	Added ZipVFS boot loader behaviors

For a prototype, suitable for introduction in a point release, it was decided that no new features would be added to the C library of the Tcl core itself. Instead, these modifications to the startup behavior are stitched into a modified Tcl shell. Modifying the Tcl library would require adding or altering entries in the Tcl stubs table. While that is fairly innocuous to hackers like me who build directly from source, it has the potential for sowing chaos for binary distributions of Tcl. It is a fairly standard practice to not alter the stubs API between, say, Tcl 8.6.2 and 8.6.3. When we get up to 8.7, that's another story entirely.

As coded, the modifications do not alter the standard Tclsh/Wish that people have come to know and love. Instead, Tcl builds two sister shells "Tclzsh(dls)" and "Tkzsh(dls)". A Tclzshd is built against a Tcl with –enable-shared=0. A Tclzshs is built against a Tcl with –enable-shared=1. Tclzshs shells are much larger, as they have the entirety of the Tcl C library statically compiled within.

There are reasons to need both types of shells. Tclzshs is a completely standalone executable. It can be transported to another machine (of a compatible architecture) and run in the absence of

any other Tcl. Tclzshd is a dynamically linked shell, just like a standard Tclsh. It simply has ZipVFS support built in, as well as an independent copy of the contents of $tclsrc/library.

On the Tk side of the world, we don't offer a statically built full-up Wish shell. Instead, we take the statically compiled Tclzshs, and embed Tk as a loadable library into its VFS. I've tried it several ways, and this was the least objectionable across both the Windows and Unix platforms.

Tclzsh/Tkzsh shells build in the same way tcltest/tktest build. Their code is actually embedded in tclAppInit.c/tkAppInit.c, but they are only activated by passing in a compile flag. In this case: TCL_ZIPVFS.

The changes are modest:

```
fossil diff -r trunk unix/tclAppInit.c
--- unix/tclAppInit.c
+++ unix/tclAppInit.c
@@ -38,11 +38,16 @@
 #ifndef MODULE_SCOPE
 #   define MODULE_SCOPE extern
 #endif
 MODULE_SCOPE int TCL_LOCAL_APPINIT(Tcl_Interp *);
 MODULE_SCOPE int main(int, char **);
+#ifdef TCL_ZIPVFS
+  MODULE_SCOPE int Tcl_Zvfs_Boot(const char *,const char *,const char *);
+  MODULE_SCOPE int Zvfs_Init(Tcl_Interp *);
+  MODULE_SCOPE int Zvfs_SafeInit(Tcl_Interp *);
+#endif /* TCL_ZIPVFS */
 /*
  * The following #if block allows you to change how Tcl finds the startup
  * script, prime the library or encoding paths, fiddle with the argv, etc.,
  * without needing to rewrite Tcl_Main()
  */
@@ -78,11 +83,17 @@
 #endif
 #ifdef TCL_LOCAL_MAIN_HOOK
     TCL_LOCAL_MAIN_HOOK(&argc, &argv);
 #endif
+#ifdef TCL_ZIPVFS
+    #define TCLKIT_INIT     "main.tcl"
+    #define TCLKIT_VFSMOUNT "/zvfs"
+    Tcl_FindExecutable(argv[0]);
+    CONST char *cp=Tcl_GetNameOfExecutable();
+    Tcl_Zvfs_Boot(cp,TCLKIT_VFSMOUNT,TCLKIT_INIT);
+#endif
     Tcl_Main(argc, argv, TCL_LOCAL_APPINIT);
     return 0;    /* Needed only to prevent compiler warning. */
 }
 /*
@@ -109,11 +120,17 @@
     Tcl_Interp *interp)   /* Interpreter for application. */
 {
     if ((Tcl_Init)(interp) == TCL_ERROR) {
  return TCL_ERROR;
     }
+#ifdef TCL_ZIPVFS
+    /* Load the ZipVfs package */
+    Tcl_StaticPackage(interp, "zvfs", Zvfs_Init, Zvfs_SafeInit);
+    if(Zvfs_Init(interp) == TCL_ERROR) {
+ return TCL_ERROR;
+    }
+#endif
 #ifdef TCL_XT_TEST
     if (Tclxttest_Init(interp) == TCL_ERROR) {
  return TCL_ERROR;
     }
 #endif
```

We introduce one major function: Tcl_Zvfs_Boot(), which is implemented in tcl/generic/tclZipVfsBoot.c. This function is called before Tcl_Main, and does the following:

1. Detect a Zip archive appended to this executable
2. If detected, mount that archive as a file system rooted at /zvfs
3. If so mounted, detect the path /zvfs/boot/tcl.
4. If detected, populate the **tcl_library** variable with that path. This will cause Tcl to look for init.tcl there instead of searching for it on its own.
5. If /zvfs was mounted, also look for /zvfs/boot/tk, and map that to **tk_library**.
6. If /zvfs was mounted, detect the presence of a **main.tcl** in the root folder
7. If /zvfs/main.tcl was detected, pass that file location to Tcl_SetStartupScript()

When the interpreter finally loads, we also provide several functions to the interpreter for managing ZipVFS mounted volumes in Zvfs_Init().

Working ~~with~~ around Zlib

The Tcl build process has a love/hate relationship with the on-board Zlib in many operating systems. On Unix, autoconf does a pretty decent job of figuring out if we can use the local Zlib, and if not building the constituent parts for Tcl. On Windows, we go so far as to not only bundle the Zlib sources, but also a binary copy of a holy golden Zlib DLL's, with hand tuned assembler and a pile of other platitudes laid on top. I think the real story is that the folks building on Microsoft Visual Studio need the version of Zlib built from MSYS to link against.

As the operating motto of this project is "Brute force and ignorance." ZipVFS enabled shells compile and link their own Zlib functions straight out of the sources provided in $tclsrc/compat/zlib. In the absolute worst case, the shell is slightly larger. But the take away is that the shell doesn't get into the discussion about whether Tcl has compiled it's own Zlib, linked to the native operating system's Zlib, or does the roundabout thing on the Windows platform with distributing it's own Zlib dll[2].

[2] No seriously, that's why you will find a zlib.dll in the c:/tcl/bin directory.

Creating a ZipVFS shell of your own.

Here is the part of our program where we break out our command lines and commence playing. For this tutorial, I will take the liberty of assuming that we are working on a development machine of the Unix-a-like persuasion, with all of the build tools to compile Tcl.

For this tutorial, we will be building an experimental copy of Tcl in ~/tmp/tcl.

```
fossil clone http://core.tcl.tk/tcl ~/Download/tcl.fos
mkdir -p ~/tmp/tcl
fossil open ~/Download/tcl.fos core_zip_vfs
cd ~/tmp/tcl/unix
./configure
make binaries
make tclzsh
```

Example 1

Note: the directions did not say to "**make install**". You can do this later if you like, but for now we want to demonstrate a few nice thing about this zip-enabled shell can do *without* having to install Tcl.

For the tutorials, a line that begins with a **$** is from the operating system command line. A line starting with **%** is run within Tcl. A line with neither a **$** or **%** is the output of a program. This was the least confusing schema I could come up with, and as these proceedings will be rendered in grayscale, color-coding won't transfer. Plus I know of at least one party interested in reading this paper who is color-blind.

The location of tcl_library

Out of the box, Tclzshd behaves exactly like a Tclsh. If a script is given as an argument, it is executes. With no command line arguments, it enters into interactive mode.

```
$./tclsh
% set tcl_library
/Users/seandeelywoods/odie/lib/tcl8.6
% exit
$ ./tclzshd
% set tcl_library
/zvfs/tcl8.6
% exit
```

Example 2

In Example 2 we see that **tclsh**, built from the same fossil checkout as our modified shell, performs a search to find an init.tcl. And, it gets that init.tcl from a semi-random location. The ZipVFS enabled **tclzshd** is using it's own copy of init.tcl.

Having achieved my stated goal (namely controlling where Tcl gets it's init), lets see what other things this new toy can do.

ZipVFS Shell Tutorial

With Tclzshd in hand, let's start:

```
$ echo puts {Hello World} > hello.tcl
$ zip -Aq tclzshd hello.tcl
$ rm hello.tcl
```

In the above example, we create a short program to a file called "hello.tcl". We pack that file into our executable with the help of the **zip** command. The –Aq argument tells **zip** to recomputed the offsets (A) and be quiet about what it's doing (q). We need the –A because we are working with a self-extracting executable. And just to make sure it's clear we are working out of the VFS, destroy the original script.

```
$ ./tclzshd
% source /zvfs/hello.tcl
Hello World
% exit
```

In the above example, we call Tclzshd with no arguments. It opens in interactive mode, and we can use the **source** command to exercise the hello.tcl script. We see our output, and exit.

```
$ /tclzshd /zvfs/hello.tcl
Hello World
$
```

In the above example, we show that the Tclzshd behaves just like a normal Tcl shell would if we pass it the name of a script to run. In this case, the script is located within its attached VFS. The reason why we can do this is because /zvfs is actually mounted before the interpreter initializes. So, to the interpreter, accessing a file in /zvfs is just the same as accessing any other file. We can also call code internal to the VFS from the command line.

```
$ echo puts [list Your arguments were {*}\$argv] > echoargv.tcl
$ zip -Aq tclzshd echoargv.tcl
$ tclzshd /zvfs/echoargv.tcl Foo bar baz
Your arguments were Foo bar baz
$ tclzshd echoargv.tcl Foo bar baz
Your arguments were Foo bar baz
```

In the above example, we demonstrate that calling scripts with command line arguments works the same way for VFS embedded scripts as normal scripts.

Adding a main.tcl

Once we introduce a "main.tcl", that tcl script will become the boot sequence for the shell. It will never again enter interactive mode. How it handles command line arguments is up to the program itself.

```
$ cp tclzshd myshell
$ echo puts [list This shell will self destruct in... [lindex ::\$argv 0]] > main.tcl
$ zip -Aq myshell main.tcl
$ ./myshell 10
This shell will self destruct in...10
$
```

In the above example:

- We made a copy of **tclzshd** called **myshell**.
- We gave **myshell** a very simple main.tcl.
- That main.tcl output something snarky to stdout.
- With no other code, Tcl got to the end of the program and exited.

In fact, this main.tcl script is now the only program **myshell** will run:

```
$ echo puts {Hello World} > hello.tcl
$ ./myshell hello.tcl
This shell will self destruct in... hello.tcl
$ ./myshell Some nonsense
This shell will self destruct...Some
```

Un-bare-able Size

There is a price to pay for freedom. (Isn't there always?) When tucking a complete copy of $tclsrc/library onto an executable, the VFS is also populated with all of Tcl's encoding tables. All of that extra data adds up to over a megabyte, even compressed:

```
$ ls -lh tcl*
-rwxr-xr-x  1 seandeelywoods  staff   1.0M Oct 19 13:28 tclzshd
-rwxr-xr-x  1 seandeelywoods  staff    13K Oct 19 13:28 tclsh
```

For this reason, the core_zip_vfs branch secretly saves a copy of the tclkit without the VFS.

```
$ cd ~/tmp/tcl/unix ; ls -lh tclkit* tclsh
-rwxr-xr-x  1 seandeelywoods  staff   1.0M Oct 19 18:53 tclzshd
-rwxr-xr-x  1 seandeelywoods  staff    30K Oct 19 18:53 tclzshd_bare
-rwxr-xr-x  1 seandeelywoods  staff    13K Oct 19 18:53 tclsh
```

The tclsh (unmodified) is about 13kb. The Zip enabled tclkit_bare, with no VFS is 30kb. The tclzshd is 1.0mb. If we don't mind Tcl doing a scavenger hunt for init.tcl, tclzshd_bare is still a fully functioning shell:

```
./tclzshd_bare
% set tcl_library
/Users/seandeelywoods/odie/lib/tcl8.6
% source hello.tcl
Hello World
% exit
```

If you want to form it into a dedicated system tool, all that is required is a copy of Zip.

```
$ cp tclzshd_bare mynewapp
$ zip main.zip main.tcl
$ cat main.zip >> mynewapp
$ zip -A mynewapp
Zip entry offsets appear to be off by 31244 bytes - correcting…
$ ./mynewapp
This shell will self destruct in... {}
$
```

Now we have a "mynewapp", and it's not that large:

```
$ ls -lh mynewapp
-rwxr-xr-x  1 seandeelywoods  staff    31K Oct 20 22:04 mynewapp
```

Building Our Own "Zip" Executable

What if we don't have a resident copy of zip? Included with the **core_zip_vfs** patch is a set of tcl routines to provide basic zip/unzip capabilities to Tcl. The routines are part of the zvfstools package, and they are bundled along with http, platform and the other core packages.

In fact, if you look through the Makefile, we never actually call zip to build Tclzshd:

```
# Builds an executable linked to the Tcl dynamic library
${TCLZSH_EXE}: ${TCLZSH_BASE}_bare  tclzsh.vfs
	@$(TCL_EXE) ../tools/mkzip.tcl ${TCLZSH_EXE} \
		-runtime ${TCLZSH_BASE}_bare \
		-directory tclzsh.vfs
	chmod a+x ${TCLZSH_EXE}
```

That ../tools/mkzip.tcl is a very short file:

```
$ cat ../tools/mkzip.tcl
###
# Wrapper to allow access to Tcl's zvfs::mkzip command from Makefiles
###
source [file join [file dirname [file normalize [info script]]] \
    .. library zvfstools zvfstools.tcl]
zvfs::mkzip {*}$argv
```

If we wanted to make a Tcl-based version of zip:

```
$ mkdir mkzip.vfs
$ cp ../library/zvfstools/zvfstools.tcl mkzip.vfs
$ echo source /zvfs/zvfstools.tcl > mkzip.vfs/main.tcl
$ echo zvfs::mkzip {*}\$argv >> mkzip.vfs/main.tcl
$ ./tclsh ../tools/mkzip.tcl mkzip -directory mkzip.vfs -runtime tclzshd_bare
$ chmod +x mkzip
```

And if we try to run it:

```
$ ./mkzip newzip.zip mkzip.vfs
$ ls -lh newzip.zip
-rw-r--r--  1 seandeelywoods  staff    22B Oct 23 06:07 newzip.zip
```

And if we call out the program with the wrong arguments, Tcl even handles the error message for us:

```
$ ./mkzip
wrong # args: should be "zvfs::mkzip filename ?arg ...?"
    while executing
"zvfs::mkzip {*}$argv"
    (file "/zvfs/main.tcl" line 2)
```

Further Work

This project is essentially in the working demo stage. It needs to address a few issues before it can move from the half-bakery and onto store shelves.

Volume Support

Currently ZipVFS uses the Unix standard behavior of taking control of a part of the file system. A neater approach would be for zvfs to mount archives in a completely different volume. Instead of /zvfs, something like exec:/ or boot:/ or zvfs0:/. In other words, give the mounted volumes a name that can, in no way, mask part of the true file system of the host machine. At least by default.

Windows (StrangelHelpful)ness

Windows tries to be extremely helpful and automatically pre-appends any paths sent to the file system resolver with the boot volume (c:) So a call to:

```
% glob /zvfs/*
```

Comes into the function that ZipVFS exports to do this lookup as “c:/zvfs”. That behavior seems wrong to me, but I’m afraid that fixing the issue has the potential to break a lot of software that is currently relying on it.

To make glob work on Windows, ZipVFS currently ignores the initial c:

While reviewing this paper, Andreas Kupries noted:

You now see the problem. An absolute path on Unix is volume relative on Windows. As Unix maintains a CWD Windows also maintains a 'current volume', which then gets added. I think that cmd.exe shows the current volume as part of the prompt.

Note that if the current volume is 'f:' etc, then you should see that getting added.

This is actually a strike for using volume-based paths (i.e. zfvs0:....) in general for the zip filesystems.

Thread Safety

ZipVFS does a few nice things to make it relatively safe to pass around in threads. It stores the table of contents of the Zipfile in cached data structures. File access, however, is not very safe. To access a file within the archive Tcl must:

- open the file channel
- seek to the start location of the embedded file
- traipse along the length of the embedded file, feeding the raw data into inflate()
- close the file channel.

Eventually two threads are going to want to access the VFS at the same time. If they get into a fighting match, the results could be confusing.

I believe I have identified the critical points where the ZipVFS code is accessing the archive file. These points have been wrapped in Tcl_MutexLock() and Tcl_MutexUnlock() calls. As written, only one thread at a time will be allows access to the IO critical portions of the ZipVFS driver. While this is acceptable for all of the envisioned use cases (namely an executable loading a common file system for all interpreters and threads) this design would have to be elaborated further if we wanted to allow, for instance, two threads to have non-contested access to two different archives.

Test Suite

To truly be a sanctioned part of the core, ZipVFS will need a test suite. It has none currently. Plenty of anecdotes from the field about how well it works, but no test suite to speak of.

Support for Code Signing

One more area that will require a deft touch is in the area of code signing. Code signing works by appending a certificate to the end of the executable. The end of the file is also the same spot that ZipVFS is looking for the table of contents. Given that the principle reason to bother with ZipVFS is to make self contained executables, the ZipVFS implementation will have to be able to detect a code signing certificate, and know to move further up the file to locate it's table of contents.

Integration with the Tcl C Library

The implementation as described is not, technically, adding Zip file support to the core. At least the core as defined as "code accessible from the tcl dynamic loadable." It is a modification to the Tcl shell that is distributed with the core. Integration with the dynamic library will have to wait for the next major release of Tcl. Modifying the stubs table on a point release is a definite no-no.

Picking a better name

"Tclzsh" does not exactly roll off the tongue. I am open to suggestions.

Conclusion

This project has been submitted as TIP #430. I urge you all to try it out.

If you have any questions, comments, or contributions, I can be reached:

- On the TkChat app as "hypnotoad"
- Via email at: yoda@etoyoc.com (For personal correspondence)
- Via email at: swoods@tnesolutions.com (For business correspondence)

Acknowledgements:

Cover art:

The official Tcl logo, Artist: Laurent Demailly, source:
http://www.tcl.tk/images/logos/TclTkLogo.html

Zipper Graphic, Artist: Laura Strickland, source:
http://content.mycutegraphics.com/graphics/letter/zipper-black-white.png

Existing Code that was Adapted for this Project

tclZvfs.c

tclZvfs.c is a hybrid of several Zvfs.c implementations in the wild.

(Circa 2000)	Richard Hipp developed the original zvfs.c file for a project called “Tcl as One Big Executable” (TOBE).
2002-01-27	Development continued by Peter MacDonald, who added support for the (then new) virtual file system hooks for the Tcl core
2006-2009	Development continued by Dennis LaBelle, who added encrypted file support, and integrated the file into the FreeWrap project.
2009-	Development continued by Sean Woods, and Dennis LaBelle (in parallel) to adapt zvfs.c to work with the new integrated Zlib in Tcl 8.6
2014-August	Sean Woods and Donal Fellows begin curatorial work on the file, removing compiler warnings, removing no longer used code.
2014-October	Sean Woods fixes glob handling on Windows, adds thread mutexes.

ziptools.tcl

The encoder for ziptools is adapted from Pat Thoyts implementation on the wiki, posted to: http://wiki.tcl.tk/15158. The encoder in ziptools has been modified to also create records in the table of contents for directories, as well as files. The boot loader it generates for kit files now looks for a pkgIndex.tcl file as well as a main.tcl file.

Additional Thanks

Many thanks are also due to Donald Porter, Donal Fellows, and Andreas Kupries. Their input in many areas helped guide the final form of this project. And their knowledge of all things Tcl saved me a lot of trial and error. I would also like to thank Roy Keene. He was always eager to crack open his code to compare implementations, which also helped tremendously.

Also thanks to Andreas Kupries and Will Duquette for proofreading the early drafts of this paper.

Appendix

Package zvfstools

zvfstools is a pure-tcl package that is distributed with the other core packages (http, package, etc.) It is accessible after calling:

```
package require zvfstools
```

zvfs::mkzip *archive args*

Creates a zip archive in 'filename'. If a file already exists it will be overwritten by a new file.
This command takes the following options:

-comment *string*	Provide a comment for the archive
-directory *path*	When given, the new zip archive will be rooted in the provided directory. If not specified, the tool mimics the behavior of **zip**, and uses the current working directory.
-exclude *patternlist*	Exclude the specified patterns from the search for files to add to the archive. Default: `{CVS/* */CVS/* *~ ".#*" "*/.#*"}`
-runtime *filename*	Use *filename* as a self-executable prefix for this archive
-zipkit	If specified, a preamble will be added to the archive to make it suitable for loading into Tcl with the *source* command.

zvfs::unzip *archive path*

This command unpacks the file system of the zipfile *archive* into the directory specified in *path*. This implementation exploits the **zvfs::mount** capabilities of the new shell, and essentially does a recursive file copy.

Package zvfs

A ZipVFS enabled shell contains a static package called zvfs. This package provides the following commands to the interpreter:

zvfs::mount ?archive? ?mountpoint?

With 2 arguments, it mounts the zipfile *archive* to the mount point *mountpoint*.
With 1 argument: it returns the mount point of *archive*, or throws an error if *archive* is not mounted.
With no arguments it returns a list of all archives and their mount points.

zvfs::unmount archive

Unmounts the zipfile *archive*