CLIF FLYNT

the tclsh spot

CREATING STAND-ALONE EXECUTABLES WITH TCL/TK

Clif Flynt is president of Noumena Corp., which offers training and consulting services for Tcl/Tk and Internet applications. He is the author of *Tcl/Tk: A Developer's Guide* and the TclTutor instruction package. He has been programming computers since 1970 and a Tcl advocate since 1994.

clif@cflynt.com

DYNAMIC LANGUAGES LIKE TCL ARE great for rapid development. In a few hours you can churn out applications that would take days or weeks (or even months) to develop in compiled languages like C, C++, or Java.

The downside of developing an application in a high-level dynamic language is that clients need to have the appropriate interpreters and libraries installed on their system before they can run your application.

The solution to this problem is to wrap the application and interpreter into a single executable. For Tcl, there are several choices:

- Tcl-Wrapper (http://sourceforge.net/projects/tclpro/): The first Tcl wrapper, it was developed by Scriptics as part of the TclPro development suite. This is the only wrapping application that stores the Tcl code as compiled bytecodes, providing some code obfuscation. This application is now supported by ActiveState (http://www.activestate.com) as part of the TclDev package.
- Wrap (http://www.xs4all.nl/~nijtmans/wrap.html): Jan Nijtmans created a proof-of-concept wrapping application to demonstrate how a smaller executable could be made using upx and wrap. Jan is no longer supporting this application, but it spawned the current StarPack, Freewrap, and TOBE wrappers.
- FreeWrap (http://freewrap.sourceforge.net/): This application is written by Dennis LaBelle and is based on D. Richard Hipp's mkTclApp. This is excellent for pure Tcl applications. The command-line interface is very simple and clean.
- StarPack (http://www.equi4.com/): Developed by Jean-Claude Wippler and Steve Landers, this is part of a set of deployment solutions that includes a single-file Tcl/Tk interpreter, compressed applications to be run by that single-file interpreter, and wrapped executables with interpreter, application code, and data. This package has more features than FreeWrap. It is very useful for wrapping a pure Tcl application or one that includes a Tcl-stubs-enabled library. It uses a more complex application build sequence, which may take a few steps to create an application.
- TOBE (http://www.hwaci.com), developed by D. Richard Hipp, provides the most control and supports extensions that are not Tcl-stubs-enabled. Using this package requires compiling a small "C" code wrapper and linking that to the Tcl libraries.

All of these wrapping solutions are built around a zip archive. Part of the zip-file specification allows a prefix to be placed ahead of the actual archive. The prefix can even be an executable program, which allows a zip archive to be an application. This is how self-extracting zip files are created. The Tcl wrapping programs all use a modified tclsh or wish interpreter as the prefix.

The problem with just prepending the tclsh interpreter to a zip file is that a useful Tcl interpreter is not just a single executable file. When the Tcl interpreter starts it loads a number of Tcl files with support for other commands.

In order to make a single executable, these files need to be included with the archive, and the Tcl interpreter needs to understand how to find the files.

Enter Tcl's Virtual File System (VFS) API to solve the problem. Just as UNIX streams generalized the interface between different devices, Tcl's VFS generalizes the interface between different directory systems. With a little bit of glue to read a directory format, any collection of files can be mounted as a directory and the files can be read. If the collection supports writing, they can also be written.

For example, the VFS API enables a Tcl script to mount an FTP site as a directory, search the site with the glob and cd commands, and open and read files with open, gets, and read commands.

For a wrapped application, this means that the Tcl support libraries can be placed in the zip archive and the Tcl interpreter can be told to look for them there, instead of looking for them on the hard drive (/usr/local/lib/tcl8.4, for instance).

The default search path for the Tcl libraries is compiled into the Tcl interpreter. You can define an alternative path to the libraries with the environment variables TCL_LIBRARY and TK_LIBRARY.

The basic steps in creating a wrappable Tcl interpreter are:

- Write glue functions to interface between the Tcl VFS and the file collection format.
- Write a function that will:
  1. Mount the file collection.
  2. Set TCL_LIBRARY to point to the new directory.
  3. Set TK_LIBRARY to point to the new directory.
  4. Initialize Tcl and Tk interpreters.
  5. Load and evaluate the Tcl application.
- Add code to invoke your Tcl initialization procedure.

Any of the wrappers described above will work for a pure Tcl application. To wrap a FORTRAN application, however, we need more control. The big problem is that a FORTRAN application will have a main entry point defined by the FORTRAN compiler/linker. This will conflict with the main function in a normal tclsh.

The TOBE paradigm lets us put the code to initialize the Tcl interpreter in a function that is invoked by the FORTRAN code, rather than in the normal main function.

When you download TOBE from http://www.hwaci.com/sw/tobe/index.html you get:

- Sample Makefile for Linux and cross-compiling with mingw.
- zvfs.c, the glue that lets a zip file be accessed like a file system.
- tkwinico.c, a function that provides a custom windows icon.

- main.c, the main function that mounts the zip file and initializes the Tcl interpreter.
- main.tcl, a sample Tcl application to wrap.

I'll start with a simple example of using TOBE to wrap a single script, and then show how to use TOBE to wrap a FORTRAN application.

The first step is to be certain you have Tcl and Tk static libraries available. Many distributions only include the dynamic libraries, but to make a completely self-contained executable, we need to link with static libraries.

If you don't have libtcl*.a on your system, you can build it.

To build Tcl from scratch:

1. Download the Tcl sources from http://sourceforge.net/projects/tcl/.
2. Untar the archives (tar -xvzof tcl8.4.6-src.tar.gz).
3. Change to the UNIX directory (cd tcl8.4.6/unix).
4. Configure the Makefile (./configure -disable-shared).
5. Make the libraries (make).
6. Repeat for tk.

Note that you need to include the -disable-shared option to configure. The default configure script makes shared libraries, but not static libs.

Also, you don't need to install the new libraries. We can build TOBE applications using a different version of Tcl than the default on our system.

The next step to use TOBE for a simple application is to edit the Makefile. The sample Makefile is created with hard links to Richard Hipp's home directories, and is guaranteed not to work for anyone else. However, the Makefile includes commented-out generic paths as examples of what might exist on your system.

Use your favorite editor to find each of the /drh/ lines in the Makefile and comment them out. Either uncomment a previous generic line or define a path that's appropriate to your system.

My preference is to place the TOBE and application directories in the same directory as the Tcl and Tk source directories, and to use relative paths in this format:

```
###### The linker option used to link against the TCL library
#
LIB_TCL = ../tcl8.4.6/unix/libtcl8.4.a -lm -ldl
```

The default Makefile has hooks for many Tcl extensions, including BLT, SQLite, and Img. The executable will be smaller if we don't include extensions we aren't using, so uncomment all of the -DWITHOUT_foo options in the Makefile.

Once this is done, you should be able to type make in the tobe directory, watch it build zvfs.o, main.o, etc., link these with the Tcl and Tk libraries, tack a little bit of zip magic onto the end of the file, and build an executable zip file.

Any errors indicate that you don't have the paths set correctly or are missing a -L option in a library path definition.

When this is done, you should be able to use unzip -t to examine the contents of the new file and confirm that it really is a zip file.

The sample main application in main.c is hardcoded to run the Tcl program main.tcl in the zip archive.

You can add code for a simple Tcl application to src/main.tcl, rerun make, and create a new tobe that will run that application. For real projects, modify the Makefile to generate an application with the name you prefer, or just rename the tobe executable this creates.

To use TOBE with the FORTRAN/Tcl library described in the previous couple of "Tclsh Spot" articles, we need to merge code from the TOBE main.c into ftcl_start (in ftcl_c.c) to initialize the interpreter from the zip archive, instead of using the default files located on your system.

The original ftcl_start function took a single argument, the name of the script to load. The modified version requires two arguments: the name of a script to load and the name of the executable. (Your FORTRAN program can get this information using the f2kcli package from http://www.winteracter.com/f2kcli/index.htm.)

We need to pass the name of the executable to Tcl_FindExecutable after creating the Tcl interpreter. The Tcl_FindExecutable function finds the full path to the application and saves it internally for use by a number of the Tcl interpreter's housekeeping tasks.

After creating the Tcl interpreter with

```
ftcl_interp = Tcl_CreateInterp();
Tcl_FindExecutable(executableName);
```

the code can set a few global variables. For this application, we aren't accepting any command-line flags, so the argv and argc global variables are set to empty and 0, respectively. The argv0 variable holds the name of the application, which is stored locally in an array named executableName. Finally, the tcl_interactive variable is set to false to let the Tcl interpreter know that it's running a script, not running as an interactive shell.

When tclsh is used as an interactive shell, the Tcl interpreter tries to evaluate each line as a Tcl command, and if that fails, it tries to evaluate the line as a system command. This is proper for an interactive shell, but inappropriate behavior for most scripts.

```
Tcl_SetVar(ftcl_interp, "argv", "", TCL_GLOBAL_ONLY);
Tcl_SetVar(ftcl_interp, "argc", "0", TCL_GLOBAL_ONLY);
Tcl_SetVar(ftcl_interp, "argv0", executableName,
        TCL_GLOBAL_ONLY);
Tcl_SetVar(ftcl_interp, "tcl_interactive", "0", TCL_GLOBAL_ONLY);
```

Next, the zip file system is mounted and Tcl's global environment array is set to point to the zip file system.

Tcl keeps a copy of the user's environment in the env array. All of the environment variables you can set in your shell are stored in this array, with the environment variable name used as the array index. For instance, puts $env(PATH) would print out the path.

In this case, since we need to force the Tcl interpreter to look for the initialization files in the zip archive, we overwrite the original values for the TCL_LIBRARY and TK_LIBRARY indices to point to the zip file system.

```
/* We have to initialize the virtual file system before calling
** Tcl_Init(). Otherwise, Tcl_Init() will not be able to find
** its startup script files.
*/

/* Initialize the zip file system package */
Zvfs_Init(ftcl_interp);

/* Mount the zip archive (this executable) as /zvfs */
retval = Zvfs_Mount(ftcl_interp, Tcl_GetNameOfExecutable(),
        "/zvfs");

/* Point env(TCL_LIBRARY) and env(TK_LIBRARY) to the zip direc-
        tories */
Tcl_SetVar2(ftcl_interp, "env", "TCL_LIBRARY", "/zvfs/tcl",
        TCL_GLOBAL_ONLY);
```

```
Tcl_SetVar2(ftcl_interp, "env", "TK_LIBRARY", "/zvfs/tk",
        TCL_GLOBAL_ONLY);
```

Now the code can initialize the Tcl and Tk interpreters with Tcl_Init and Tk_Init:

```
if( Tcl_Init(ftcl_interp) ) {
    ftcl_debug_message( "ftcl_start - Tcl_Init:",
    Tcl_GetVar(ftcl_interp, "errorInfo", TCL_GLOBAL_ONLY)) ;
    return 1;
}
if( Tk_Init(ftcl_interp) ) {
    ftcl_debug_message( "ftcl_start - Tk_Init:",
    Tcl_GetVar(ftcl_interp, "errorInfo", TCL_GLOBAL_ONLY)) ;
    return 1;
}
```

If your application might need to create new interpreters with extensions loaded, you must include a call to Tcl_StaticPackage to let the Tcl interpreter know that a statically linked package has been loaded into the interpreter.

For example, if you need to create child interpreters with Tk loaded, you might do it with these commands. Note that the load is invoked with an empty string and a package name, instead of the more common usage of providing a file name. This format is used when the file is already loaded and the script just needs to invoke the extension's initialization function in the new slave interpreter.

```
interp create withwish
withwish eval {load "" tk}
withwish eval {pack [canvas .c]}
withwish eval {.c create text 100 50 -text "child interp"}
```

If you leave out the Tcl_StaticPackage call, this code would generate an error message like

```
package "tk" isn't loaded statically
```

By including this line, the Tk package can be loaded into a new slave interpreter.

```
Tcl_StaticPackage(ftcl_interp,"Tk", Tk_Init, 0);
```

If your application has other extensions it may need to load into slave interpreters, you must include a call to Tcl_StaticPackage for each one.

We need to make a couple of modifications to the FORTRAN and Tcl code to run inside a TOBE application.

The code we had in lander.f90 called ftcl_start with the single argument config.tcl. The ftcl_start function then called Tcl_EvalFile to load and evaluate the config.tcl script in the current directory.

When we package and ship this application, we'll have a single file, and there won't be any config.tcl in the current directory. The config.tcl file will be in the zip archive, which is mounted as /zvfs.

Changing the original ftcl_start from this original:

```
CALL ftcl_start('config.tcl')
```

to this provides the application name and causes the Tcl_EvalFile to try to evaluate the file in the zip archive:

```
CHARACTER(256) :: exe
CALL get_command_argument(0,exe)
CALL ftcl_start('/zvfs/config.tcl', exe)
```

Similarly, in the config.tcl script that loads the GUI and starts the Lunar Lander application running, we originally just sourced files from the current directory like this:

```
source options.tcl
```

```
source setConditions.tcl
source GUI6.tcl
```

In order to run within a TOBE, we need to source these files from the /zvfs directory. The simple solution is to just add the /zvfs/ to the paths.

```
source /zvfs/options.tcl
source /zvfs/setConditions.tcl
source /zvfs/GUI6.tcl
```

The final steps are to compile the new ftcl_c.c, link the application, and create the magic TOBE zip archive.

These steps are all in the sample Makefile that comes with TOBE. They can be combined into a single sequence of commands like this:

```
lander: lander.f90 ftclz.a
    $(FC) -o tobe.zip $(FLAGS) lander.f90 ftclz.a $(LIBS)
    rm -rf zipdir
    mkdir zipdir
    ln -s /usr/local/lib/tcl8.4 zipdir/tcl
    ln -s /usr/local/lib/tk8.4 zipdir/tk
    cp $(TCL_APPFILES) zipdir
    cat ../tobe/src/null.zip >>tobe.zip
    cd zipdir; /usr/bin/zip -qr ../tobe.zip *
    cd ..
    mv tobe.zip lander
```

And with that, we've created a stand-alone Tcl/FORTRAN application that can be shipped to our client without worrying whether they have the proper Tcl interpreters installed on their system.

As usual, the complete code described in this article is available at http://www.noucorp.com.