`libtclsh.a`: A linkable library for embedding Tcl

Clif Flynt
Noumena Corporation
9300 Fleming Rd
Dexter, MI 48130
`clif@noucorp.com`
`www.noucorp.com`

Introduction

One of Tcl's strengths is that it can be embedded into another application to provide that application with new features. The problem with this is that an application with Tcl embedded into it will also need a complete Tcl distribution in order to use these features. Functionality like that offered by like http package, some Tk widgets and the tools implemented in the Tcllib are only available if they can be loaded at run time.

There are several wrapping solutions for creating a single-file executable from a Tcl script. FreeWrap, ProWrap, Tobe, and StarKits are the best known of these. These solutions use the Tcl Virtual File System (VFS) to include the ancilliary files with the executable.

This paper introduces a technique for making a single library file that can be linked to a compiled application creating a single file executable that includes the ancilliary libraries.

Problem

The immediate problem that needed to be solved was that our group was presented with a FORTRAN program that needed IP Socket support, and might also need graphics support. The developers of the FORTRAN application were not familiar with Tcl and were resistant to adding another learning curve to their development schedule.

By providing them with a single file to link with their application, the Fortran to Tcl library developed by Arjen Markus, and a Tcl script that we provided, we were able to create an application very quickly that had the mathematical models they had developed and the interactive capabilities that we developed.

Solution

This solution is composed of 5 parts, each of which extends the functionality of the previous parts.

1. The `bzip2` library (`bzip2-1.0.3`).
2. An Tcl extension that compresses and decompresses bzipped data (`bzTcl`).
3. A VFS Extension that can access a memory resident bzipped filesystem (`memFile`).
4. A modified main.c and Tcl_Interp creation procedure (`memTclsh`).
5. The FTCL package from Arjen Markus with hooks to the new library (`FORTRAN_Tclsh`).

`bzip2`

The `bzip2` library is taken from the net and not modified. For this iteration, I used the 1.0.3 release, downloaded from `http://www.bzip.org`.

`bzTcl`

The `bzTcl` extension provides an interface to the `bzip2` library. This extension was originally developed to read compressed file data from Project Gutenberg. It was extended to include pure memory compression and decompression.

The functions supported are:

- compress in memory data.
- decompress in memory data.

- read compressed data from a file and decompress it.
- write compressed data to a file from plaintext memory.

`memFile`

The `memFile` extension creates a memory based Virtual File System and provides the hooks for the Tcl interpreter to access this. This set of code includes functionality for reading and searching a VFS, but does not implement writing new code to in-memory VFS.

`memTclsh`

The `memTclsh` section of this package is derived from D. Richard Hipp's TOBE (Tcl as One Big Executable) main and CreateInterp functions, which are derived from the normal Tcl main and CreateInterp functions. The main function is provided to build a single-file tclsh (or wish) interpreter that needs no ancilliary files. This is similar to a standalone TOBE or Starkit. The `memCreateInterp` function duplicates the behavior of the normal Tcl_Main, creating the new interpreter, setting the initial global variables, initializing the filesystem, setting up the package library paths and loading init.tcl, etc.

`FORTRAN_Tclsh`

The `FORTRAN_Tclsh` library is Arjen Markus's FTCL package modified slightly to create a memTcl interpreter instead of a standard Tcl_CreateInterp interpreter.

Details

The bulk of the code that implements this system is the VFS support functions.

The memory file system is constructed as an array of `memFileData` structs:

```
typedef struct memFileData {
  char *name;           // Path to file, rooted at the "filesystem" root.
  int dataLen;          // Number of bytes in data.
  unsigned char *data;  // BZ2 Compressed file data.
} memFileData;
```

The filesystem itself is an array of these structures. Since the filesystem will be relatively small (about 1000 entries), I decided to opt for simplicity rather than any sort of optimized access.

The Tcl Virtual File System is created by filling in the fields of a `Tcl_Filesystem` structure and registering the filesystem with the `Tcl_FSRegister` command.

The Tcl_Filesystem structure is implemented as a structure containing a set of pointers to procedures which implement file system functions. It is defined in the `tcl.h` and looks like this:

```
/*
 * struct Tcl_Filesystem:
 *
 * One such structure exists for each type (kind) of filesystem. It collects
 * together in one place all the functions that are part of the specific
 * filesystem. Tcl always accesses the filesystem through one of these
 * structures.
 *
 * Not all entries need be non-NULL; any which are NULL are simply ignored.
 * However, a complete filesystem should provide all of these functions. The
 * explanations in the structure show the importance of each function.
 */

typedef struct Tcl_Filesystem {
    CONST char *typeName;     /* The name of the filesystem. */
    int structureLength;      /* Length of this structure, so future binary
                               * compatibility can be assured. */
    Tcl_FSVersion version;      /* Version of the filesystem type. */
    Tcl_FSPathInFilesystemProc *pathInFilesystemProc;
                                /* Function to check whether a path is in this
                                 * filesystem. This is the most important
                                 * filesystem function. */
    Tcl_FSDupInternalRepProc *dupInternalRepProc;
                                /* Function to duplicate internal fs rep. May
                                 * be NULL (but then fs is less efficient). */
    Tcl_FSFreeInternalRepProc *freeInternalRepProc;
                                /* Function to free internal fs rep. Must be
                                 * implemented if internal representations
                                 * need freeing, otherwise it can be NULL. */
    Tcl_FSInternalToNormalizedProc *internalToNormalizedProc;
                                /* Function to convert internal representation
                                 * to a normalized path. Only required if the
                                 * fs creates pure path objects with no
                                 * string/path representation. */
    Tcl_FSCreateInternalRepProc *createInternalRepProc;
                                /* Function to create a filesystem-specific
                                 * internal representation. May be NULL if
                                 * paths have no internal representation, or
                                 * if the Tcl_FSPathInFilesystemProc for this
                                 * filesystem always immediately creates an
                                 * internal representation for paths it
                                 * accepts. */
    Tcl_FSNormalizePathProc *normalizePathProc;
                                /* Function to normalize a path.  Should be
                                 * implemented for all filesystems which can
                                 * have multiple string representations for
                                 * the same path object. */
    Tcl_FSFilesystemPathTypeProc *filesystemPathTypeProc;
                                /* Function to determine the type of a path in
                                 * this filesystem. May be NULL. */
    Tcl_FSFilesystemSeparatorProc *filesystemSeparatorProc;
                                /* Function to return the separator
                                 * character(s) for this filesystem. Must be
                                 * implemented. */
    Tcl_FSStatProc *statProc;     /* Function to process a 'Tcl_FSStat()' call.
                                 * Must be implemented for any reasonable
                                 * filesystem. */
    Tcl_FSAccessProc *accessProc;
                                /* Function to process a 'Tcl_FSAccess()'
                                 * call. Must be implemented for any
                                 * reasonable filesystem. */
    Tcl_FSOpenFileChannelProc *openFileChannelProc;
                                /* Function to process a
                                 * 'Tcl_FSOpenFileChannel()' call. Must be
                                 * implemented for any reasonable
                                 * filesystem. */
    Tcl_FSMatchInDirectoryProc *matchInDirectoryProc;
                                /* Function to process a
                                 * 'Tcl_FSMatchInDirectory()'.  If not
                                 * implemented, then glob and recursive copy
                                 * functionality will be lacking in the
                                 * filesystem. */
    Tcl_FSUtimeProc *utimeProc;    /* Function to process a 'Tcl_FSUtime()' call.
                                 * Required to allow setting (not reading) of
                                 * times with 'file mtime', 'file atime' and
                                 * the open-r/open-w/fcopy implementation of
                                 * 'file copy'. */
    Tcl_FSLinkProc *linkProc;     /* Function to process a 'Tcl_FSLink()' call.
                                 * Should be implemented only if the
                                 * filesystem supports links (reading or
                                 * creating). */
    Tcl_FSListVolumesProc *listVolumesProc;
                                /* Function to list any filesystem volumes
```

```
                              * added by this filesystem. Should be
                              * implemented only if the filesystem adds
                              * volumes at the head of the filesystem. */
    Tcl_FSFileAttrStringsProc *fileAttrStringsProc;
                             /* Function to list all attributes strings
                              * which are valid for this filesystem. If not
                              * implemented the filesystem will not support
                              * the 'file attributes' command. This allows
                              * arbitrary additional information to be
                              * attached to files in the filesystem. */
    Tcl_FSFileAttrsGetProc *fileAttrsGetProc;
                             /* Function to process a
                              * 'Tcl_FSFileAttrsGet()' call, used by 'file
                              * attributes'. */
    Tcl_FSFileAttrsSetProc *fileAttrsSetProc;
                             /* Function to process a
                              * 'Tcl_FSFileAttrsSet()' call, used by 'file
                              * attributes'.  */
    Tcl_FSCreateDirectoryProc *createDirectoryProc;
                             /* Function to process a
                              * 'Tcl_FSCreateDirectory()' call. Should be
                              * implemented unless the FS is read-only. */
    Tcl_FSRemoveDirectoryProc *removeDirectoryProc;
                             /* Function to process a
                              * 'Tcl_FSRemoveDirectory()' call. Should be
                              * implemented unless the FS is read-only. */
    Tcl_FSDeleteFileProc *deleteFileProc;
                             /* Function to process a 'Tcl_FSDeleteFile()'
                              * call. Should be implemented unless the FS
                              * is read-only. */
    Tcl_FSCopyFileProc *copyFileProc;
                             /* Function to process a 'Tcl_FSCopyFile()'
                              * call. If not implemented Tcl will fall back
                              * on open-r, open-w and fcopy as a copying
                              * mechanism, for copying actions initiated in
                              * Tcl (not C). */
    Tcl_FSRenameFileProc *renameFileProc;
                             /* Function to process a 'Tcl_FSRenameFile()'
                              * call. If not implemented, Tcl will fall
                              * back on a copy and delete mechanism, for
                              * rename actions initiated in Tcl (not C). */
    Tcl_FSCopyDirectoryProc *copyDirectoryProc;
                             /* Function to process a
                              * 'Tcl_FSCopyDirectory()' call. If not
                              * implemented, Tcl will fall back on a
                              * recursive create-dir, file copy mechanism,
                              * for copying actions initiated in Tcl (not
                              * C). */
    Tcl_FSLstatProc *lstatProc;    /* Function to process a 'Tcl_FSLstat()' call.
                              * If not implemented, Tcl will attempt to use
                              * the 'statProc' defined above instead. */
    Tcl_FSLoadFileProc *loadFileProc;
                             /* Function to process a 'Tcl_FSLoadFile()'
                              * call. If not implemented, Tcl will fall
                              * back on a copy to native-temp followed by a
                              * Tcl_FSLoadFile on that temporary copy. */
    Tcl_FSGetCwdProc *getCwdProc;
                             /* Function to process a 'Tcl_FSGetCwd()'
                              * call. Most filesystems need not implement
                              * this. It will usually only be called once,
                              * if 'getcwd' is called before 'chdir'. May
                              * be NULL. */
    Tcl_FSChdirProc *chdirProc;    /* Function to process a 'Tcl_FSChdir()' call.
                              * If filesystems do not implement this, it
                              * will be emulated by a series of directory
                              * access checks. Otherwise, virtual
```

```
                                       * filesystems which do implement it need only
                                       * respond with a positive return result if
                                       * the dirName is a valid directory in their
                                       * filesystem. They need not remember the
                                       * result, since that will be automatically
                                       * remembered for use by GetCwd. Real
                                       * filesystems should carry out the correct
                                       * action (i.e. call the correct system
                                       * 'chdir' api). If not implemented, then 'cd'
                                       * and 'pwd' will fail inside the
                                       * filesystem. */
} Tcl_Filesystem;
```

As the comments note, sections of this structure may be blank if that functionality is not present. For example, you don't need write procedures on a read only file system. Other procedures will improve efficiency, but are not required for base functionality.

The Tcl_Filesystem structure defined for this extension resembles this:

```
Tcl_Filesystem memFileSystem = {
    "memFile",
    sizeof(Tcl_Filesystem),
    TCL_FILESYSTEM_VERSION_1,
    &memFile_PathInFilesystem,
    &memFile_DupInternalRep,
    &memFile_FreeInternalRep,
    NULL,
    NULL,
    NULL,
    &memFile_FilesystemPathType,
    &memFile_FileSystemSeparator,
    &memFile_Stat,
    &memFile_Access,
    &memFile_OpenFileChannel,
    &memFile_MatchInDirectory,
    &memFile_Utime,
    NULL,
    &memFile_ListVolumes,
    &memFile_FileAttrStrings,
    &memFile_FileAttrGet,
    &memFile_FileAttrSet,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL};
```

These are the critical functions for a read only file system, and how they were implemented.

```
    Tcl_FSPathInFilesystemProc *pathInFilesystemProc;
/*  Function to check whether a path is in this filesystem.
   This is the most important filesystem function. */
```

The memory file system is the structure defined above. The memory filesystem keeps a Tcl hash table for to define the mounted memory file systems. The key for these is teh mount point, and the associated value is the address of the memory structure.

To simplify life, the memory file system can only be mounted at the toplevel. Thus to determine whether a path is on the file system, the code can strip the path to the parent directory, then check to see if that is a key for a mounted file system with `Tcl_ FindHashEntry`

```
    Tcl_FSFilesystemPathTypeProc *filesystemPathTypeProc;
/* Function to determine the type of a path in this filesystem. May be
NULL. */
```

The only type of entity in this file system is a file. There is no distinction between directories and files because there are no directory names, just file names that have many `foo/bar` type strings in them.

The body of this function looks like this:

```
  retObj = Tcl_NewStringObj("memFile", -1);
  Tcl_IncrRefCount(retObj);
  return retObj;
```

```
    Tcl_FSFilesystemSeparatorProc *filesystemSeparatorProc;
/* Function to return the separator character(s) for this filesystem.
Must be implemented. */
```

This can also be implemented as a hardcoded value.

```
  retObj = Tcl_NewStringObj("/", -1);
  Tcl_IncrRefCount(retObj);
  return retObj;
```

```
    Tcl_FSStatProc *statProc;
/* Function to process a 'Tcl_FSStat()' call. Must be implemented for
any reasonable filesystem. */
```

If this procedure simply returns TCL_OK, everything seems to work.

It should fill the Tcl_StatBuf structure that's passed to the function. Many of these fields are irrelevant to a read-only file system.

```
    Tcl_FSAccessProc *accessProc;
/* Function to process a 'Tcl_FSAccess()' call. Must be implemented for
any reasonable filesystem. */
```

This function will return TCL_OK if the path is in the file system and can be accessed and -1 if the file does not exist. Since there are no permission bits on this filesystem, any file that exists can be accessed. The implementation is that the HashEntry for the mount point is first acquired. The data value associated with this is the address of the array of memory File structurs. This array is searched to see if there is a file with the requested name in the array.

Note that -1 is not TCL_ERROR, however, `Tcl_FSAccess` in `tclIOUtil.c` returns a -1 and the code that invokes it will also expect -1 for failure.

```
    Tcl_FSOpenFileChannelProc *openFileChannelProc;
/* Function to process a 'Tcl_FSOpenFileChannel()' call. Must be
implemented for any reasonable filesystem. */
```

Opening a new channel is well described in the Tcl man pages and Welch's *Practical Programming in Tcl/Tk*.

When a file is opened, a new Tcl_Channel structure must be created using the `Tcl_CreateChannel` function.

One of the arguments to `Tcl_CreateChannel` is a pointer *Instance* structure that this file system uses to hold information about the open file.

On a normal file system this will include a read buffer, a start pointer, length, etc.

For this memory filesystem, the instance data includes a buffer that will hold the uncompressed file data, length and offset fields. The functions listed in the `Tcl_ChannelType` structure use these values to copy data from the memory file buffer to the Tcl memory area.

```
    Tcl_FSMatchInDirectoryProc *matchInDirectoryProc;
/*  Function to process a 'Tcl_FSMatchInDirectory()'.  If not
implemented, then glob and recursive copy functionality will be lacking
in the filesystem. */
```

This function is the trickiest of the required functions. While this is not required for a VFS, it is required by the initialization scripts that use various flavors of `glob` to find the runtime loaded files and packages.

There are many permutations used by the `glob` command, thus this function might be invoked with a Tcl_GlobTypeData pointer which contains data or may be NULL, a pattern which may also be NULL and path which may or may not be rooted.

This function includes too much code to turn the path and pattern into a generic format. After the string is generalized, the `Tcl_ StringCaseMatch` command can be used to determine if the pattern matches the names of the files in the memFile structure.

Using `Tcl_StringCaseMatch` saved writing a great deal of pattern matching code.

```
    Tcl_FSListVolumesProc *listVolumesProc;
/*  Function to list any filesystem volumes added by this filesystem.
Should be implemented only if the filesystem adds volumes at the head
of the filesystem. */
```

The code associated with this function uses the `Tcl_NextHashEntry` to step through the hash table of mounted file systems and returns the keys that are in use.

The filesystem array is constructed as a c code source file by a Tcl script. The script recursively descends from a root directory, compresses the appropriate files and outputs an ascii equivalent of the binary codes.

The C code resembles this:

```
// /usr/local/lib/tcl8.5/msgs/zh_hk.msg
static unsigned char mem_data0[] = {66,90,1...};
memFileData file0 = {"/tcl8.5/msgs/zh_hk.msg", 347, mem_data0 };
// /usr/local/lib/tcl8.5/msgs/zh_sg.msg
static unsigned char mem_data1[] = {66,90,1...};
memFileData file1 = {"/tcl8.5/msgs/zh_sg.msg", 255, mem_data1 };
...
memFileData *tclTkInitData[] = {
&file0,
&file1,
&file2,
...};
```

This technique of embedding Tcl code in a C memory structure was borrowed with some modifications from the SafeTcl extension developed by Marshall Rose and Nathaniel Borenstein.

This method of building large data structures in 3 steps is supported by both newer and older GCC compilers.

The C code that's generated by this application Tcl initialization library is about 4000 lines long (5,628,135 bytes, compiling down to 1,781,828 bytes), and the Tk initialization library is about 3555 lines lon (compiling down to 1,278,272).

Once the `libtclsh.a` or `libwish.a` has been created, it can be linked with a C application.

A trivial test looks like this:

```
#ifdef USE_TK
#include <tk.h>
#else
#include <tcl.h>
#endif
#include <ctype.h>
#include <errno.h>
#include <string.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdlib.h>

#include "memFile.h"

int main(int argc, char **argv){
  Tcl_Interp *interp;
  interp = memCreateInterp(argc, argv);
  Tcl_Eval(interp, "puts OK");
  return 0;
}
```

It can be compiled with:

```
(reverse-i-search)`gc': gcc test1.c libtclsh.a -lm -ldl
```

Lessons Learned

1. Use the Source, Luke! The documentation for the VFS system is not as complete as it might be. In places, it isn't quite intuitive (such as the Access function that returns -1 instead of a TCL_ERROR when it fails).
2. The Tcl and Tk initialization code provides a fairly complete suite to exercise a VFS file system.
3. There are an amazing number of permutations to handling file globbing.

Future Work

There are many FS functions that are dummied out which could be included to improve efficiency or add functionality.

One goal for this method of embedding Tcl in applications is to be able to use C code to easily invoke Tk widgets and map callbacks from a Tk Widget back into C code. This has the potential for making Tk a useful tool for prototyping new Widgets that a C developer might use.

The current version of this package is working and in use under Linux using gcc 3.3 and gcc 4.0. Porting the make environment to Windows and Macintosh has been considered and might happen.

The code is available at `www.dedasys.com/~clif`.