freeWrap

6.41

Documention

Build stand-alone TCL/TK executables. No compiler required!

OR

Use it as a single-file WISH shell

freeWrap 6.41 Documentation

Table of Contents

freeWrap 6.41 Documentation

freeWrap License

freeWrap 6.41 Documentation

Overview

The freewrap program turns TCL/TK scripts into single-file binary executable programs.

The resulting program can be distributed to machines that do not have TCL/TK installed. The executable will also work on machines that have TCL/TK installed but will use its own TCL/TK "image". freeWrap itself does not need TCL/TK installed to run.

Easy, one-step wrapping.
FreeWrap consists of a single executable file. There is no setup required. Wrapping is accomplished with a single command.

Your source and data files are protected from prying eyes.
FreeWrap automatically encrypts all files you wrap into your executable application to provide a secure distribution.

freewrapTCLSH can be used to wrap TCL-only scripts.
FreewrapTCLSH creates a single executable file from a TCL script. The wrapping syntax is identical to the freewrap program. This produces a console-only type of program.

freeWrap can be used as a single file stand-alone WISH
Renaming the freeWrap program to some other file name causes freeWrap to behave as a a stand-alone, single-file WISH that can be used to run any TCL/TK script or a freeWrap package containing all the files in your application.

freewrapTCLSH can be used as a single file stand-alone TCLSH shell
Renaming the freewrapTCLSH program to some other file name causes freewrapTCLSH to behave as a a stand-alone, single-file TCLSH shell that can be used to run any TCL script or a freeWrap package containing all the files in your application.

Shared libraries can be used with your wrapped programs.
FreeWrapped applications can load TCL/TK shared library extensions that have been compiled with the STUBS interface.

Your wrapped programs can be customized with your own window icons.
The Windows version of freeWrap can incorporate your own customized icon into your wrapped application.

No license fees for wrapped programs.
There are no license fees associated with freeWrap. See the freeWrap license .

Cross-platform generation of programs is supported.
The -w "wrap using" option allows cross-platform creation of wrapped applications without the use of the target computer system.

freeWrap includes several Windows-specific commands
These commands can be used to determine the location of Windows' special directories and make for easy creation of file extension associations and shortcuts.

freeWrap includes commands for ZIP file creation and extraction.
Due to freeWrap's use of the ZIP Virtual File System any ZIP archive can be opened so its contents look like a simple file subdirectory. The archive's files are automatically decompressed when read with TCL

commands.

The makeZIP command allows creation and modification of ZIP archives from within your freeWrapped application.

freeWrap 6.41 is based on TCL/TK version 8.5.1.

Availability

FreeWrap executables are freely available for both Linux and Windows95/98/NT/2000/XP. Instructions and source code for building freeWrap on both Windows and UNIX platforms are included in the freeWrap source code distribution.

Versions of freeWrap that include the SQLite and TkTable extensions to TCL/TK are also available for download.

TCL-only versions of freeWrap are also available for wrapping TCL (non-TK) scripts.

Visit http://sourceforge.net/projects/freewrap to download files.

freeWrap 6.41 Documentation

freeWrap as a TCL/TK wrapper program

FreeWrap can wrap TCL/TK applications that consist of multiple script and binary image files. FreeWrap combines all the files together into a single executable file. The syntax for wrapping an application is described below.

Calling Syntax:	freewrap mydir/prog.tcl [-debug] [-f FileLoadList] [-forcewrap] [-i ICOfile] [-o OutFile] [-p] [-w WrapStub] File1 ... FileN	
where:	mydir/prog.tcl	file path to main TCL/TK program script
	File1 ... FileN	A list of space-separated text or binary files to include in the wrapped application.
	-debug	Opens a console window so user can see debug messages while wrapping.
	-f	Specifies that the following named file (FileLoadList) contains a list of files to wrap
	-forcewrap	Force freeWrap to act as a wrapping program even if it has been renamed.
	-i	Substitute the following named Windows ICO file (ICOfile) as the program application icon.
	-o	Indicates that the name of the produced executable program should be set to OutFile.
	-p	Create a freeWrap program package instead of an executable program.
	-w	Specifies that the following named file (WrapStub) is the name of the file to use as the freeWrap stub
output:	prog (Linux) prog.exe (Windows) **Note**: the output file will be placed in the current directory from which freeWrap is called.	

The names of the files being wrapped may include either relative or full paths. The resulting executable program can access the wrapped files by either refering to them by their full path as they existed at the time of wrapping or adding the paths to TCL's ***auto_path*** variable. If the ***auto_path*** method is used, the appropriate *tclIndex* or *pkgIndex.tcl* files should also be wrapped into the application. Please see the information on how to wrap a package extension. Please see Naming and referring to wrapped files for more details on how to refer to wrapped files within the application.

Both text and binary files can be wrapped.

-debug Option

Use of the -debug option will display the freeWrap console window so any debug warning messages can be viewed while wrapping.

-f Option

For larger wrap projects, the user may wish to use freeWrap's -f option to specify a file which contains a list of files to wrap. The specified text file must contain one file name per line. Each file name listed in the file will be added to the wrapping. Use of the -f option does not preclude the specification of individual files on the freeWrap command line. The -f option may also be used several times on the same command line.

Example: *freewrap myprog.tcl logo.gif -f projlist.txt code2.tcl -f special.txt*

-forcewrap Option

The -forcewrap option can be used to force freeWrap to act as a wrapping program even if it has been renamed. Without this command line option, freeWrap behaves like a WISH shell when it has been renamed.

-i Option

Use the -i option to specify the icon you wish to use for your wrapped application. This option is only relevant when wrapping an application for the Windows operating system.

Example: *freewrap myprog.tcl -i myprog.ico*

This icon must be an ICO formatted file. This option will replace the freeWrap icon with the contents of the specified ICO file. When creating your ICO file, keep in mind that freeWrap contains four versions of the freeWrap icon. They are:

1. 16x16 16 colors
2. 32x32 16 colors
3. 32x32 2 colors
4. 32x32 256 colors

Icons, of these resolutions, found in your ICO file will be used to replace the freeWrap icons. If your ICO file doesn't contain at least these four versions of your icon then only the matching icons will be replaced. This would leave a freeWrap icon that could be displayed by Windows at some time.

-o Option

The -o Option allows you to specify the name of the executable program you are creating/wrapping.

-p Option

Using the -p option creates a wrapped application without the freeWrap executable component. This file is called a freeWrap program package. A freeWrap program package can be run using freeWrap as a single-file shell By default, freeWrap program packages are given a file extension of *fwp*.

Example wrapping: freewrap myapp.tcl -p

Example execution: freewish myapp.fwp

-w Option

By default, freeWrap attaches the wrapped files to a copy of the freeWrap program you use to do the wrapping. The -w option allows attaching the wrapped files to a different copy of freeWrap. Since freeWrap is available for multiple operating systems, this feature is useful for assembling freeWrapped applications for other operating systems while on a single computer.

Example (assembling a Windows version while running freeWrap on Linux):

freewrap myprog.tcl -w freewrap.exe

Example (assembling a Linux version while running freeWrap on Windows):

freewrap myprog.tcl -w freewrap

Remember, the argument following the -w option must be the file path to a version of the freeWrap program that can execute on the other operating system.

Wrapping already wrapped files

Even files previously generated by freeWrap can be wrapped into another freeWrap application. However, the repetitive inclusion of freeWrap's TCL/TK code would produce fairly large application files. Therefore, the freeWrap program has been designed to provide efficient packaging of previously wrapped applications. FreeWrap removes a wrapped program's freeWrap core prior to storing it in the wrapped application. Only the application's archive section is stored in the new freeWrap application. This archive section is given a name starting with the string fwpkg_ followed by its original file name and having an extension of ZIP . It is the responsibility of the programmer to use the ::freewrap::reconnect command to later reattach the freeWrap core and copy the full application to a disk file.

Example (assume firstApp is a freeWrap generated application)

freewrap myprog.tcl firstApp.exe

creates a file named fwpkg_firstApp.zip inside the application myprog.exe. To restore and copy the original application (firstApp.exe) to disk the myprog.exe program should use a TCL command similar to:

::freewrap::reconnect fwpkg_firstApp.zip c:/myprog/bin/firstApp.exe

freeWrap as a single-file WISH interpreter

Renaming the freeWrap program to some other file name causes freeWrap to behave as a a stand-alone, single-file WISH that can be used to run any TCL/TK script or freeWrap program package. This can be done in the following manner.

Copy freewrap.exe to a new file name

Example: *copy freewrap.exe wishrun.exe*

Use the new file as you would normally use WISH

Example: *wishrun script_name.tcl*

freeWrap program packages

FreeWrap normally produces an executable file when wrapping an application. However, it is also possible the create a file that only contains the wrapped files for the application. This allows you to distribute smaller packages that can later be run using freeWrap as a single-file TCLSH or WISH interpreter. A freeWrap program package can contain all the files for your application in a single compressed file.

Use the -p option when wrapping your application in order to create a freeWrap program package instead of an executable file.

Example wrapping: freewrap -f listOfiles.txt myapp.tcl -p

Example execution: freewish myapp.fwp

FreeWrap program packages are not encrypted and can be run using a copy of freeWrap 6.3 or later..

freeWrap's console window

Under freeWrap, the ***console*** command is available for both Windows and UNIX. The console window is the location that will receive any STDOUT or STDERR output. The console can also be used to interactively enter TCL/TK commands. Use ***console show*** to display the window and ***console hide*** to remove it.

Using the DDE and Registry packages (Windows only)

The DDE and Registry packages have been compiled into freeWrap. There is no need to load them with a ***package require*** command. Simply use the ***dde*** and ***registry*** commands without any preceding ***package require*** command.

Using wrapped files.

Wrapping

When running a wrapped application, the first file specified on the command line at the time of wrapping will be executed as a TCL/TK script. All other files specified on the command line or in a file load list are available to this executing script.

You CAN do the following with the wrapped files.

1. ***Source*** them
2. ***Open*** them
3. ***Read*** them
4. ***Close*** them
5. ***Glob*** them
6. Use any ***file*** commands that do not write to the files
7. Use them with the ***image create*** command
8. Specify them for ***-bitmap*** widget options.

You CANNOT do the following with the wrapped files.

1. ***File delete*** them (since they exist in the application, not on disk)
2. Use the ***load*** command on them. However, you can write them to disk first then use the ***load*** command on the new disk file. You may, instead, consider providing any ***load***-able extension as a separate file instead of wrapping it.

Naming and referring to wrapped files

All files included in a wrapped application must be referred to by their full path within the application. However, any relative or full path specification can be used on the freeWrap command line.

Windows users will notice that freeWrap strips all drive letter information from a file's path prior to storing it inside the wrapped application. When referenced inside the wrapped program, the path to the wrapped files must have no drive letter. To the wrapped application, all of its internal files will appear to be on the same "default" drive.

For example, if an application is wrapped to include the file C:\projects\myproject\libmodule1.tcl with the following command:

```
freewrap myapp.tcl libmodule1.tcl
```

You would need to use a source command within the application such as:

```
source /projects/myproject/libmodule1.tcl
```

DO NOT expect the relative path of wrapped files to change when you move the executable program.

FreeWrap takes a "snapshot" of the file path for all wrapped files. You must use the same, full path (minus any drive letter) that existed at the time of wrapping to refer to the wrapped file. It is also important that the file paths you use in your program exactly match the letter case that exists at the time of wrapping.

These rules also apply to the ***file*** or ***open*** commands. Also, make sure the path you add to auto_path corresponds to the wrapped tclIndex file you include in your application. For example, if your wrapping command is:

```
freewrap myapp.tcl c:\devel\myapp1\tclIndex c:\devel\myapp1\libmodule1.tcl
```

you should add /devel/myapp1 to auto_path.

In summary:

You should use the paths to the files as they exist at the time of wrapping. Wrapping takes a "snapshot" of the file path for all wrapped files. Do not use relative paths to refer to wrapped files within the application since relative paths will not be found.

Wrapping and using TCL/TK extensions (packages)

TCL/TK extensions can be wrapped into your application and then loaded dynamically at run time. Alternatively, if you are willing to recompile freeWrap, TCL/TK extensions may also be statically compiled into freeWrap. See http://sourceforge.net/projects/freewrap for versions of freeWrap that already include some statically compiled TCL/TK extensions.

Wrapped applications can load TCL/TK shared binary extension that have been compiled with the new TEA (i.e., stubs) interface. Stubs-enabled shared libraries can be included in the wrapped application or exist as separate files.

TCL's package search mechanism uses the ***glob*** command to recursively search directories specified in the auto_path variable to find packages. Unfortunately the ***glob*** command does not do the same for Virtual File System (VFS) files or their directories. This means TCL's package require command will not descend subdirectories when searching for packages. However, the fix for this is simple. Add the desired package's file path to the auto_path variable before using the package require command. This can be done with two lines of code similar to:

```
lappend auto_path /tcl/lib/mypkg1.0 ;# Ensure our app can find the files
package require mypackage
```

Script only extensions

Packages consisting only of TCL/TK scripts are generally easy to wrap.

As an example, let us consider the BWidget 1.8 extension. Under Windows the following short batch file could be used to wrap a sample program using BWidgets named BWidget_demo.tcl.

```
REM wrapBWidget.bat file
dir /S /B .\BWidget1_8 >Bwidget_files.txt
START /WAIT freewrap.exe BWidget_demo.tcl -f Bwidget_files.txt
```

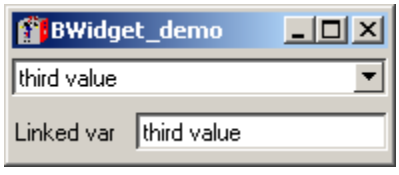
These batch commands create a text file containing the list of files that make up the extension (one file name per line). These commands also assume that the BWidget package has been installed in the BWidget1_8 directory immediately below the current directory.

For our example, the BWidget_demo.tcl file if found in the current directory and contains the following TCL/TK commands.

```
lappend ::auto_path [file dirname [zvfs::list */Bwidget1_8/pkgIndex.tcl]]
package require BWidget

set combo [ComboBox .combo \
   -textvariable comboVal \
   -values {"first value" "second value" "third value" "fourth value" "fifth value"} \
   -helptext "This is the ComboBox"]

set ent [LabelEntry .ent -label "Linked var" -labelwidth 10 -labelanchor w \
   -textvariable comboVal -editable 0 \
   -helptext "This is an Entry reflecting\nthe linked var of ComboBox"]

pack $combo $ent -pady 4 -fill x
```

The first line of this sample script allows the [package require BWidget] command to find the BWidget extension.

Running this wrapped application will produce the following pop-up window.



Extensions containing a single binary file

TCL extensions with a single binary file (other script files may be present) are also easy to wrap if:

1. The binary library does not call other binary libraries located inside the wrapped application.
2. The binary library does not call scripts located inside the wrapped application.

Your application code simply needs to add the path to the binary extension's pkgIndex.tcl file to the auto_path variable so that the usual ***package require*** or ***load*** commands will find the extension.

The **zvfs::list** command can be used to make setting auto_path easier. For example:

```
lappend ::auto_path [file dirname [zvfs::list */myExtension/pkgIndex.tcl]]
```

More complex extensions with both scripts and binary libraries

If the TCL extension contains more than one binary library or one of the libraries is dependent on other libraries wrapped into your application you will need to use something like the procedure found in the following example. This situation is rare and you may never need to do this. However, the Tclxml package is one such extension and we will use it for our example.

The Tclxml extension has several supporting scripts whose loading is normally automated by the [package require] command. We need to use the [package require] command but the associated pkgIndex.tcl file doesn't really know how to load our DLL.

This is easy to correct by making a slight modification to Tclxml's pkgIndex.tcl file so that it uses the following procedure, freeWrap_load, instead of the regular load command.

```
proc freewrap_load {libfile args} {
     # This procedure should be used instead of the normal LOAD command when
     # using binary extensions with freeWrap.
     #
     # Returns: On success, the full file path to the shared library on the
     #          local file system.
     #          On failure, an error message starting with the text "Load
     #          Error: "
     #
     global env
     set rtnval {}
     set fpath [::freewrap::unpack $libfile]
     if {[string length $fpath] == 0} {
         set rtnval "Load Error: Copying of shared library $libfile failed."
        } {
           if {[file mtime $libfile] > [file mtime $fpath]} {
               # The wrapped library file is newer than the one on disk.
               # First delete the existing one, then copy the newer file.
               if {[catch {file delete -force $fpath}]} {
                   set fpath {}
                   set rtnval {Load Error: Unable to delete older copy of
shared library.}
```

```
                    } {
                       set fpath [::freewrap::unpack $libfile]
                       if {[string length $fpath] == 0} {
                           set rtnval {Load Error: Unable to replace older copy
of shared library.}
                       }
                    }
                }
            }
        if {[string length $rtnval] == 0} {
            # No errors, so far. Let's load the shared library.
            if {[catch "load \{$fpath\} $args" result]} {
                set rtnval "Load Error: $result"
               } { set rtnval $fpath }
           }
        return $rtnval
}
```

In our situation, there is one more thing to worry about. The Tclxml.dll has some dependencies (i.e., calls other DLLs). Therefore, we need to:

1. wrap these other DLLs into our program
2. unpack these other DLLs to the local file system before loading the Tclxml DLL

Again, this is easy to accomplish by making a simple change to the Tclxml pkgIndex.tcl file that we wrap into our application.

Here is a procedure for wrapping an application containing and using Tclxml version 3.1.

1. Modify the Tclxml pkgIndex.tcl file as follows:

   Replace the following line in the pkgIndex.tcl file

   ```
   package ifneeded xml::c 3.1 [list load [file join $dir Tclxml31.dll]]
   ```

   with (the following is a single line. It may suffer from wrap around)

   ```
   package ifneeded xml::c 3.1 [foreach fpath [zvfs::list *.dll]
{freewrap::unpack $fpath}; freewrap_load [zvfs::list */Tclxml31.dll]]
   ```

   This modification will unpack all DLLs from the wrapped binary, then load the Tclxml DLL itself.

2. Add some code to the application to adjust the auto_path variable. We must ensure that all wrapped directories containing pkgIndex.tcl files are added to auto_path. For this project, the following code is added at the beginning of the application. It should be executed before the [package require] command.

   ```
   foreach fpath [zvfs::list */win/*/pkgIndex.tcl] {
               lappend auto_path [file dirname $fpath]
         }
   ```

   As you can see, with this code, the location of the files is determined at run time. We don't have to

keep track of them to properly update the code. The [zvgs::list] command is very useful for locating your wrapped files.

3. Use a [package require xml] command in the application to load the Tclxml package.
4. FreeWrap the application making sure all the supporting DLLs and script files are included.

Using the WINICO features

Windows versions of freeWrap incorporate release 0.6 of the Winico extension. The following files associated with the Winico extension are included along with the freeWrap documentation.

File	*Description*
license.winico	The Winico license included with the Winico 0.6 distribution
readme.winico	The README file included with the Winico 0.6 distribution
winico.html	The Winico version 0.6 documentation in HTML format

Special variables, procedures and commands defined by freeWrap

The ::freewrap namespace

FreeWrap has a namespace which contains all of the freeWrap specific variables, commands and procedures. These variables, commands and procedures may be referenced using the ::freewrap:: prefix or imported into any other namespace.

Variables

The following variables are defined in the ::freewrap namespace of each wrapped application.

<table>
<tr><th>Name</th><th>Description</th></tr>
<tr><td>errormsg</td><td>This variable is set by the ::freewrap::unpack procedure when a file cannot be written to the requested destination.</td></tr>
<tr><td>patchLevel</td><td>Revision level of the freeWrap program used to wrap the application.</td></tr>
<tr><td>progname</td><td>The proper name for the freeWrap program for the current operating system. This is normally freewrap.exe under Windows and freewrap under UNIX.</td></tr>
<tr><td>runMode</td><td>This variable indicates whether freeWrap is running as:
<table>
<tr><th>Value of variable</th><th>Meaning</th></tr>
<tr><td>interactiveShell</td><td>an interactive shell</td></tr>
<tr><td>programPackage</td><td>a wrapped executable program</td></tr>
<tr><td>standAloneShell</td><td>a stand-alone shell running a script</td></tr>
</table>
</td></tr>
</table>

The freeWrap stub

It is important to note that all freeWrapped applications contain a copy of the freeWrap program itself. The first part of a wrapped application consists of this freeWrap "stub". Therefore, knowing the size of this freeWrap stub, it is possible to extract freeWrap from the wrapped application and copy it to another file.

Procedures

The following procedures are defined in the ::freewrap namespace of each wrapped application. The commands names starting with shell_ are only available under the Windows operating system.

Syntax: getStubSize [execname]

Description: Retrieves the current size of the freeWrap stub associated with the file execname . This function returns the size in bytes or 0, if the stub size cannot be determined.

Syntax: isSameRev file_name

Description: Checks whether the specified file contains a copy of the same freeWrap revision as the currently executing program.

Returns: 0, if file does not contain a copy and 1, if file contains a copy.

Syntax: iswrapped file_name

Description: Determines whether the file named file_name is a freeWrapped application.

If file_name is a freeWrapped application this procedure returns a value of 1.

If file_name is NOT a freeWrapped application this procedure returns a value of 0.

Syntax: reconnect pkgName, appName

Description: Reattaches the freeWrap core to pkgName, a file included in the current freeWrapped application. The recombined freeWrapped application is copied to the file appName. pkgName must be the archive portion of a freeWrapped application.

Syntax: shell_assoc_exist extension

Description: Check whether a key exists for an extension

Example: shell_assoc_exist .txt => 1
Example: shell_assoc_exist .NEVER => 0

Syntax: shell_fileType_exist fileType

Description: Determine whether a file type exists

Example: shell_fileType_exist txtfile => 1
Example: shell_fileType_exist NEVER => 0

Syntax: shell_fileExtension_setup extension, fileType

Description: Creates a file extension and associates it with fileType.
Example: shell_fileExtension_setup .txt txtfile

Remove connection between extension and fileType
Example: shell_fileExtension_setup .txt ""

Syntax: shell_fileType_setup fileType, title

Description: Creates a file type.
Example: shell_fileType_setup txtfile "Text Document"

Syntax: shell_fileType_open fileType, openCommand

Description: Creates an open command. Sets action for double click.
Example: shell_fileType_open txtfile "C:\\WINDOWS\\NOTEPAD.EXE %1"

Syntax: shell_fileType_print fileType, printCommand

Description: Creates a print command for right mouse button menu.
Example: shell_fileType_print txtfile "C:\\WINDOWS\\NOTEPAD.EXE /p %1"}

Syntax: shell_fileType_icon fileType, icon

Description: Sets an icon for a fileType.
Example: shell_fileType_icon txtfile "C:\\WINDOWS\\SYSTEM\\shell32.dll,-152"
Example: shell_fileType_icon txtfile "C:\\mydir\\myicon.ico"

We can give a name.ico file or a dll or exe file here. If a dll or exe file is used the index for the resource inside the file must be specified

Syntax: shell_fileType_quickView fileType, quickViewCmd

Description: Sets the command to execute to perform a quick view for a fileType.
Example: shell_fileType_quickView txtfile "write.exe %1"

Syntax: shell_fileType_addAny_cmd fileType, cmdName, cmd

Adds any command you want to a fileType.
Example: shell_fileType_addAny_cmd scrfile config "%1"

Syntax: shell_fileType_setMenuName fileType, cmdName, str

Description: Change description in right mouse menu for a command associated with a fileType.
Example: shell_fileType_setMenuName txtfile print "Print file"

Syntax: shell_fileType_showExt fileType, yesOrNo

Description: Always show the extension on the fileType.
Example: shell_fileType_showExt txtfile 1

Turn off "Always show" of extension on the fileType
Example: shell_fileType_showExt txtfile 0

Syntax: shell_fileType_setCmdOrder fileType, cmds

Description: Over-ride the default ordering of commands on right mouse menu.
Example: shell_fileType_setCmdOrder txtfile {print open}

Syntax: shell_fileType_neverShowExt fileType, yesOrNo

Description: Never show extension on fileType.
Example: shell_fileType_neverShowExt txtfile 1

Turn off "Never show" of extension on the fileType.
Example: shell_fileType_neverShowExt txtfile 0

Syntax: shell_getCmds file

Description: Retrieves all the commands assocated with an extension.
Example: shell_getCmds file.txt => open print

Syntax: shell_getCmd_imp file, cmd

Description: Retrieves the implimentation of a command given a file extension and command.
Example: shell_getCmd_imp test.txt open => C:\\WINDOWS\\NOTEPAD.EXE %1

Syntax: unpack file_name, [destdir]

Description: This function unpacks file_name from a freeWrapped application's ZVFS archive into a file system directory. The destination directory for the file may be specified with the optional destdir argument. If this optional argument is not specified, the function will select a temporary directory appropriate for the operating system. Unpack, on success, returns the full path to the newly created file and on failure, an empty string. This

function is useful for creating local copies of wrapped shared libraries (e.g. DLLs) that can then be loaded into your wrapped TCL/TK application.

Commands

The following TCL commands are defined in the ::freewrap namespace of each wrapped application.

Syntax: getSpecialDir dirType

Description: Find "Start Menu", "Desktop" and similar directory locations under Windows. DirType must be one of the following strings:

ALTSTARTUP	FONTS
APPDATA	HISTORY
BITBUCKET	INTERNET
COMMON_ALTSTARTUP	INTERNET_CACH
COMMON_DESKTOPDIRECTORY	NETHOOD
COMMON_FAVORITES	NETWORK
COMMON_PROGRAMS	PERSONAL
COMMON_STARTMENU	PRINTERS
COMMON_STARTUP	PRINTHOOD
CONTROLS	PROGRAMS
COOKIES	RECENT
DESKTOP	SENDTO
DESKTOPDIRECTORY	STARTMENU
DRIVES	STARTUP
FAVORITES	TEMPLATES

Syntax: makeZIP [-options] [-b path] [-t mmddyyyy] [-n suffixes] [ZIPfile FileList] [-xi list]

Description: This command duplicates most of the functionality of the Info-Zip application using the following syntax which is almost identical to the ZIP command line program.

The default action is to add or replace files in ZIPfile with the files specified by FileList.

The following optional arguments may be used.

-0	store only
-1	compress faster
-9	compress better
-A	adjust self-extracting exe
-b path	use this directory path for the temporary file
-d	delete enetries in zipfile
-D	do not add directory entries
-f	freshen: only changed files
-F	fix zipfile (-FF try harder)
-i list	include only the following file names
-j	junk (don't record) directory names
-J	junk zipfile prefix(unzipsfx)
-l	convert LF to CR LF (-ll CR LF to LF)
-m	move into zipfile (delete files)
-n suffixes	don't compress these suffixes

-o	make zipfile as old as latest entry
-r	recurse into directories
-t mmddyyyy	exclude files earlier than the specified date
-u	update: only changed or new files
-x list	exclude the following names
-X	exclude extra file attributes
-y	store symbolic links as the link instead of the referenced file

Syntax: shortcut linkPath[-objectPath objectPath] [-description description] [-workingDirectory dir] [-icon path index] [-arguments args]

Description: Creates a Windows shortcut. The only required parameter is the linkPath. This means you can create a shortcut with no target, which probably isn't useful. The icon of the shortcut will default to the icon of the target item if not specified.

Argument	Explanation
linkPath	The path to the shortcut file (including the extension .lnk)
objectPath	The target of the link
description	Shortcut description
workingDirectory	Working (startup) directory for the target of the shortcut
path index (icon)	Specifies the path to a file and the index of the icon in that file to use for the shortcut
args	Arguments passed to the target of the shortcut when started.

Character encodings

All encoding files available with the TCL distribution have been compiled into the freeWrap application.

Under the Windows version of freeWrap, the ***encoding names*** command is not able to list your wrapped encoding files. However, the other ***encoding*** commands will work correctly. As an alternative, you could use the ***zvfs::list*** command to find the wrapped encoding files. For example:

```
set encFiles [zvfs::list /tcl/encoding/*]
```

The TCL encoding files included with freeWrap can be accessed by specifying their full ZVFS path when using one of the encoding commands. For example:

```
encoding system /tcl/encoding/cp850
```

FreeWrap sets the initial system encoding to cp1252. The Tcl **source** command always reads files using the system encoding. A difficulty arises when distributing scripts internationally, as you don't necessarily know what the system encoding will be. Fortunately, most common character encodings include the standard 7-bit ASCII characters as a subset. Therefore, you are usually safe if your script contains only 7-bit ASCII characters.

If you need to use a character set other than cp1252 for the scripts that you distribute, you can provide a small "bootstrap" script written in 7-bit ASCII. The bootstrap script should first set the system encoding to the desired value then **source** the desired script.

For example, the contents of the bootstrap script (named myprogram.tcl here) could be:

```
# Set the desired encoding first.
encoding system /tcl/encoding/cp850
# Now, let's run the real program.
source myrealprogram.tcl
```

You would then wrap your application using the following command:

```
freewrap myprogram.tcl myrealprogram.tcl
```

Stdin/Stdout redirection with the exec and open commands

The Windows version of freeWrap now includes the *tcl84pip.dll* file from the normal TCL distribution. This file is necessary when running scripts that redirect stdin/stdout through pipes using the ***open*** or ***exec*** commands. *Tcl84pip.dll* is actually an executable program that helps do the redirection of stdin and stdout when ***exec***ing programs under Windows.

If your application will be using the ***exec*** or ***open*** commands to run external applications that will be piping stdin or stdout to transfer information, make sure that *tclpip84.dll* can be found in a location that TCL normally looks (e.g, the current directory, the Windows directory or the same directory as the executable program).

How freeWrap encryption works

FreeWrap now includes ZIP 2.0 style file encryption. FreeWrap encrypts all files stored into its internal

ZIP Virtual File System. It also encrypts all files wrapped into an a single-file executable application. The password required for handling the encrypted files is embedded into the wrapped program.

The password key is included as a function. Embedding the password as a compiled function makes it extremely difficult to deduce the password by looking at the executable file.

FreeWrap can easily be built to use a different embedded password that no one else has access to. This is important for generating wrapped applications where you wish to prevent others from viewing the source code. Remember, due to freeWrap's ability to mount ZIP files as a subdirectory, a person having the same version of freeWrap (with the same password) can easily read the encrypted files within your application. Therefore, those people interesting in securing the files that make up their application should use a copy of freeWrap (built with its own unique password) that no one else has access to.

FreeWrap program packages, however, are not encrypted since doing so would require distribution of the copy of freeWrap used to create the package in order to run them. Such a copy of freeWrap could them be easily used to defeat the encryption of the wrapped package.

freeWrap 6.41 Documentation

Building freeWrap

FreeWrap does not include any alterations to the TCL/TK core. It is plain-vanilla TCL/TK with sometimes an extension such as TkTable thrown in.

Dependencies

Compilation of the freeWrap executables requires the following additional libraries:

1. TCL (Tool Command Language) static library
2. TK (Tool Kit for TCL) static library
3. Zlib compression static library
4. Info-ZIP compiled object files
5. TkTable extension static library (for freeWrapPLUS version only)
6. SQLite extension static library (for freeWrapPLUS version only)

The following helper applications are needed as part of the freeWrap build process and will be called by the Make files. These programs should be placed in freeWrap's unix and win directories.

1. tclsh (used to run the setinfo.tcl and shrink.tcl scripts)
2. info-ZIP (used to attach zip archive to end of freeWrap program)

How to build freeWrap version 6.41

1. Compile static libraries for TCL and for TK

   Under Windows

   Static TCL and TK libraries can be built using the OPTS=static option with makefile.vc.

   ```
   Example:   nmake makefile.vc OPTS=static
   ```

   Under Linux

   Static TCL/TK libraries can be built under UNIX by using the --disable-shared option of the configure script. The LIB_TCL variable in makefile.linux must point to this file. This file is usually stored someplace like /usr/local/lib/libtcl85.a when "make install" is run as part of the TCL build process.

2. Compile the zlib general purpose data compression library. Version 1.2.3 was used for freeWrap 6.41.

   Source code for this library is available from the **zlib** home page: http://www.gzip.org/zlib/

3. Obtain the ZIP-2.3 program source code from Info-Zip's SourceForge site:
   http://sourceforge.net/project/showfiles.php?group_id=118012

   Do not use any other version of the ZIP program (e.g., 2.31 or 2.32). FreeWrap includes a customized version of ZIP's zip.c module that will only work properly with version ZIP 2.3.

   Compile the ZIP program. The freeWrap Make file uses the object modules produced as a result of building the ZIP program. These modules provide the ZIP program features within freeWrap. The ZIPOBJDIR variable defined in the freeWrap Make files will need to be set to the name of the directory in which the ZIP program distribution has been extracted and compiled.

   Make sure that the version of the ZIP source code you retrieve includes the encryption feature. That is, the resulting executable file has a **-e** option as part of its command line syntax.

   The freeWrap Make file also uses the ZIP program to help assemble the freeWrap executable. You should, therefore, place the ZIP program in the *unix* or *win* subdirectory (as appropriate) of the freeWrap build tree.

4. If you wish to produce the freewrapPLUS version, you will also need to obtain and compile the following TCL/TK extensions.

   TkTable 2.9 from http://sourceforge.net/project/showfiles.php?group_id=11464

   Under Linux

   The TkTable make file should be generated using the --disable-shared option of the configure script. This will cause the TkTable Make file to produce a static library file that the freeWrap Make file will use.

   Under Windows

   Use the *makefile.vc* file that comes with TkTable to build the Windows DLL. Although this method builds a DLL the freeWrap Make file will, instead, use the object files created as a result of using *makefile.vc*.

   SQLite 3.5.3 from http://www.sqlite.org/download.html

   Obtain the amalgamated version of the the SQLite source code (file sqlite-amalgamation-3_5_3.zip). Extract the files from this archive.

   Obtain the source code distribution (file sqlite-source-3_5_3.zip). Append the file tclsqlite.c found in the source code distribution to the file sqlite3.c found in the amalgamation. Compile the append file as suggested below. Remember to modify the freeWrap Make files so that LIB_SQLITE points to this compiled object file.

   Under Linux

   Compile the appended sqlite3.c using a command such as:

cc -c sqlite3.c

Under Windows

Compile the appended sqlist3.c using a command such as:

cl -DSTATIC_BUILD=1 -D__WIN32 -DWIN32 -DWINDOWS -DNDEBUG /MT /GX -Z 7 -Od /FD /c sqlite3.c

5. Use the *main.c* and other source code files provided by freeWrap.

   Use the *generic/main.c* file provided with the freeWrap source code distribution to build any version of freeWrap. This file has been written to use the ZIP Virtual File System and perform initializations normally done by the standard TCL/TK distribution. The freeWrap Make file has been designed to use *main.c*.

   The *main.c* source code is also the location in which any statically linked TCL/TK extensions should be initialized. The file currently contains examples of source code for initializing several common extensions.

   The *generic/freelib.c* and *generic/freewrapCmds.tcl* files implement some TCL commands added by freeWrap.

   The *generic/zipmain.c* file provides the code to implement the ::freewrap::makeZIP command.

   The *generic/zvfs.c* file implements the ZIP Virtual File System used by freeWrap.

   The file *generic/fwcrypt.c* will be automatically generated by the freeWrap Make file the first time freeWrap is built. This file provides a function that returns the password freeWrap will use for encryption. A new randomly selected password is generated whenever this file is recreated.

   The *generic/seticon.tcl*, *generic/setinfo.tcl* and *generic/shrink.tcl* scripts are used by the freeWrap Makefile in the creation of the freeWrap program.

   The *generic/freewrap.tcl* script is the portion of freeWrap that controls the wrapping process.

6. Use the Make file that comes with the freeWrap source code distribution to build freeWrap. You will need to modify the Make file to reflect the file paths for your computer system.

   To build different versions of freeWrap (e.g., freeWrapPLUS), modify the Make file to have the proper value of FW_EXT. See the comments in the Make file for details.

   Under Linux

   ```
   The Linux Make file is written to use the gcc compiler.

   Make command:    make -f makefile.linux
   ```

   Under Windows

   ```
   The Windows Make file is written to support the MS Visual C++
   ```

```
2008 Express Edition.
Make command:    nmake -f makefile.vc
```

8. With the addition of ZIP 2.0 style file encryption, the freeWrap build process now includes an interactive step. During this step, you must enter the password key that is compiled into freeWrap at a console prompt. This password is automatically generated the first time freeWrap is built and will be printed to the screen immediately before you need to type it in.

   The password key is included into freeWrap as a function. The source code for this function is automatically generated by the Make file the first time freeWrap is built. This source code is placed in a file named *fwcrypt.c*. Embedding the password into freeWrap as a function makes it extremely difficult to detect the password by looking at the executable file.

   Each generation of this file results in a completely different password. This allows you to build your own version of freeWrap with its own password that no one else knows. This is important for generating wrapped applications where you wish to prevent others from viewing the source code. Due to freeWrap's ability to mount ZIP files as a subdirectory, a person having the same version of freeWrap (with the same password) can easily read the encrypted files within your application. Therefore, those people interesting in securing the files that make up their application should compile their own copy of freeWrap.

ZVFS: The ZIP Virtual File System TCL Extension

Introduction

The freeWrap program is a TCL/TK script that has been attached to a single-file version of the WISH shell program. The single-file WISH was created with the help of the ZIP Virtual File System (ZVFS) source code provided by D. Richard Hipp. The ZVFS code has been adapted for use with TCL's virtual file system interface.

ZVFS is an extension to TCL that causes TCL to view the contents of a ZIP archive as real, uncompressed, individually-accessible files. Using ZVFS, you "mount" a ZIP archive on a directory of your filesystem. Thereafter, all of the contents of the ZIP archive appear to be files contained within the directory on which the ZIP file is mounted. The ZVFS extension is written in the C language.

For example, suppose you have a ZIP archive named **example1.zip** and suppose this archive contains three files named **abc.tcl**, **pqrs.gif**, and **xyz.tcl**. You can mount this ZIP archive as follows:

```
zvfs::mount  example1.zip  /zip1
```

After executing the above command, the contents of the ZIP archive appear to be files in the **/zip1** directory. So, for instance, you can now execute commands like these:

```
source /zip1/abc.tcl
image create photo img1 -data /zip1/pqrs.gif
puts "The size of file xyz.tcl is [file size /zip1/xyz.tcl]"
```

The files **/zip1/abc.tcl**, **/zip1/pqrs.gif**, and **/zip1.xyz.tcl** never really exist as separate files on your disk drive. They are always contained within the ZIP archive and are not unpacked. The ZVFS extension intercepts Tcl's attempt to open and read these files and substitutes data from the ZIP archive that is extracted and decompressed on the fly.

Using ZVFS

The ZVFS has been compiled into freeWrap using TCL's Virtual File System (VFS) interface. This extension provides the following new TCL commands:

- **zvfs::mount** ZIP-archive-name mount-point
- **zvfs::unmount** ZIP-archive-name
- **zvfs::exists** filename
- **zvfs::info** filename
- **zvfs::list** ?(**-glob**|**-regexp**)? ?pattern?

As discussed above, the **zvfs::mount** command mounts a new ZIP archive file so that the contents of the archive appear to TCLl to be regular files. The first argument is the name of the ZIP archive file. The second argument is the name of the directory that will appear to hold the contents of the ZIP archive. The ZIP archive may be unmounted using the **zvfs::unmount** command.

The **zvfs::exists** command checks to see if the file named as its first argument exists in a mounted ZIP

archive. You can do almost the same thing with the built-in **file exists** command of TCL. The **file exists** command will return true if the named file is contained in a mounted ZIP archive. But **file exists** will also return true if its argument is a real file on the disk, whereas **zvfs::exists** will only return true if the argument is contain in a mounted ZIP archive.

The **zvfs::info** command takes a single argument which is the name of a file contained in a mounted ZIP archive. If the argument is something other than such a file, this routine returns an empty string. IF the argument is a file in a ZIP archive, then this routine returns the following information about that file:

- The name of the ZIP archive that contains the file
- The uncompressed size of the file in bytes
- The compressed size of the file in bytes
- The offset of the beginning of the file in the ZIP archive

The **zvfs::list** command returns a list of all files contained within all mounted ZIP archives. If a single argument is given, that argument is interpreted as a glob pattern and only files that match that glob pattern will match. If the **-regexp** switch appears then the argument is interpreted as a regular expression and only files that match the regular expression are listed.

Limitations

The files in a ZIP archive are read-only. You cannot open a ZVFS mounted file for writing.

The renaming or deletion of files in the ZVFS is not supported.

You can **source** scripts that are ZVFS mounted files. But you cannot **load** new TCL extensions out of ZVFS. If you store a shared library or DLL in a ZIP archive, you'll have to first copy the DLL out of the archive into a real disk file before attempting to **load** it. You can use normal TCL **file copy** command to copy the DLL out of ZVFS into a real disk file and then **load** the copy, if you like. You just can not load the DLL directly out of ZVFS. This is an operating system limitation.

Overlays

ZVFS allows you to mount a ZIP archive on top of an existing file system. TCL first looks for the file in the ZIP archive and if it is not found there it then looks in the underlying file system. You can also mount multiple ZIP archives on top of one another. The ZIP archives are searched from the most recently mounted back to the least recently mounted.

This overlay behavior is useful for distributing patches or updates to a large program. Suppose you have a large application that contains many TCL scripts which you distribute as a single ZIP archive file. You can start up your application using code like the following:

```
foreach file [lsort -dictionary [glob appcode*.zip]] {
  zvfs::mount $file /appcode
}
```

This loop finds all ZIP archive (in a certain directory) that begin with the prefix **appcode**. It then mounts each ZIP archive on the same **/appcode** directory.

You can use this scheme to ship the TCL scripts of your application in a file named **appcode000.zip**. If there is later a change or update to your program that effects a small subset of the TCL scripts, you can

create a patch file named **appcode001.zip** that contains only the scripts that changed. By placing **appcode001.zip** in the same directory as **appcode000.zip** and restarting the application, all the files in **appcode001.zip** will override files with the same name in **appcode000.zip**. Subsequent updates can be named **appcode002.zip**, **appcode003.zip**, and so forth.

This kind of update scheme makes it very easy to back out a change. Suppose after trying out a particular update, the user decides they do not like it and want to go back to the prior version. All they have to do is remove (or rename) the appropriate **appcode*.zip** file and restart the application and the code automatically reverts to its previous configuration. Updates are completely and trivially reversible!

Using The Executable As The ZIP Archive

The directory information for most executable formats is at the beginning of the file and the directory information for the ZIP archive format is at the end of the file. This means that you can append extra data to an executable and the operating system will not care and you can add information to the start of a ZIP archive and the ZVFS extension will not care. So then, there is nothing to prevent you from appending the ZIP archive to the executable that contains a TCL interpreter and thereby put your entire application into a single standalone file. The freeWrap application does this.

FreeWrap is a compiled C program that creates a TCL interpreter, adds the ZVFS extention, reads the TCL initialization scripts from the attached ZIP archive then executes a TCL script from the same archive. The capabilities of the ZIP archiver program Info-ZIP has been compiled into freeWrap. These capabilities are used by freeWrap to perform all file additions and deletions to the archive portion of freeWrapped applications.

When you execute freeWrap or freeWrapped applications, the operating system loads and runs the first part of the file as the executable. Then the freeWrap code calls the ZVFS extension to read the TCL scripts from the end of the file.

ZVFS source code

The source to the ZVFS extension is contained in a single C file named **zvfs.c** and is included with the freeWrap source code distribution.