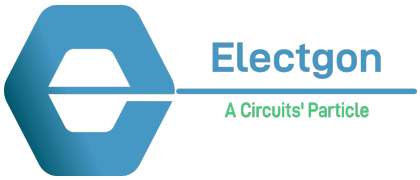

Wrapping TCL Scripts

Tclpot User Guide

ELECTGON
www.electgon.com
ma_ext@gmx.net

14.11.2020

Contents

Abstract

Although TCL is well used in many SW tools and applications, however it maybe somehow exhausting for a beginner to convert a TCL script into executable file or as it known as wrapping TCL script. There are already some free tools that can achieve this step, but it may not be satisfying as the generated executable from these tools was about 16 Mbytes of size and not easy to tweak these tools to customize your wrapper, for example changing the icon of the generated executable is a challenging task. A developer who is trying to overcome these points, he will need to tweak the free tools to fulfill his needs which will lead him to the fact that he can produce his wrapping code to exactly fit his preferences. This is what this tutorial is trying to show, how to build your own TCL wrapper. Nevertheless, the resulted wrapper can be obtained for free from tclpot repository (www.github.com/electgon/tclpot) so that you can use tclpot utility directly, just adjust associated Makefile according to your script and settings. Therefore this tutorial is discussing both the wrapping background and usage guide of tclpot source. This tutorial then contains two chapters; first chapter is discussing how to build your own wrapper. At the same time, it is explaining C source code of tclpot.c. Second chapter is showing how to run tclpot or how to adjust Makefile to build your executable.

Chapter 1

Wrapping Background

1.1 Introduction

Wrapping TCL Script means to embed TCL script in another SW routine so that we can build the later to be executed as standalone application. C/C++ is the basic programming language, so wrapping TCL means to embed TCL script into C/C++ code. TCL already provides some C API functions that can be used as interface or boundary between TCL and C. A list of available C API functions can be reviewed here [8] in which you can find that these API start mostly with TCL_..... Therefore we will use the term "TCL_C" to denote for a C application that is wrapping TCL script inside.

This wrapping attempt was inspired by two free tools called "mktclapp" and "Tobe" which seem to be quite old as it doesn't fit with new TCL distributions. This was one obstacle that hinders from using these tools directly. Although these tools are quite efficient, but they need couple of fixes in order to make it work. Actually, writing your own wrapper based on the approach of these tool will be much easier. But to explain how to understand "mktclapp"/"Tobe" or building your wrapper, we need to have the TCL source files built in the developing machine. If not, it is yet a better choice to build it yourself so that you can have the choice to build your wrapper statically or shared. We will start by this step.

1.2 Building TCL Sources

You can download source files of TCL interpreter from link referenced in [1]. In this tutorial, tcl8.7a3 has been used. So download any TCL version later than 8.7 as there is one added feature called Zipfs which we need to use. After downloading the source, extract the downloaded compressed file. You can notice that the source file after extraction will show specific directories to use while building for Linux, Windows, Mac.

1.2.1 Building in Windows

To build TCL for Windows you will need Microsoft Visual Studio framework. It is free and can be downloaded from Microsoft web page [4] or from other available online pages. We will not go here through installation steps for MS Visual Studio but this tool should be ready as we need to use its compiler for compiling both the TCL source code and other TCL_C application source codes.

To compile TCL source code simply open a “Developer Command Prompt” of Visual Studio. Then navigate to the directory that contains TCL source files then issue the following commands

```
nmake -f makefile.vc INSTALLDIR=path_to_your_install_dir
nmake -f makefile.vc install INSTALLDIR=path_to_your_install_dir
```

These previous two commands will build the TCL with shared library .dll file. To build TCL with static library use the following commands instead:

```
nmake -f makefile.vc OPTS=static,nomsvcrt INSTALLDIR=
   path_to_your_install_dir
nmake -f makefile.vc install OPTS=static,nomsvcrt INSTALLDIR=
   path_to_your_install_dir
```

Note that it is enough to use options “OPTS=static” only to build the static TCL but the option ’nomsvcrt’ is used for later purpose in compiling the TCL_C application. Upon successful build, your build directory should contain “bin”, “include”, “lib” directories. “bin” should contain tclsh.exe which you can use to open a TCL console. In case of static build, you can use this tclsh executable to run in other Windows machines. “include” directory contains some C header files that are used while building the C wrapper for TCL. “lib” directory contains compiled TCL libraries in addition to some TCL packages required for running proper TCL.

1.2.2 Building in Linux

To compile TCL for Linux, in extracted TCL source directory navigate to ’unix’ folder then use build TCL simply using:

```
./configure --prefix=path_to_your_install_dir --enable-shared=no
make
make install
```

Note here that we used the option “–enable-shared=no” to build a static library of TCL. If you want to have shared library, don’t use this option.

1.3 Building Tk Sources

If you intent to wrap a TCL script that uses Tk library, you have to prepare Tk source also. The same way we can build Tk sources, download the compressed source, then extract it to build it. But note that in order to build Tk, you must have TCL is built first. If you built TCL with option ’static’, you have to build Tk also with the same option.

1.3.1 In Windows

open uncompressed Tk source directory, then navigate to win directory and use the following commands.

```
nmake -f makefile.vc OPTS=static,nomsvcrt INSTALLDIR=
   path_to_your_install_dir TCLDIR=path_to_tcl_source
```

```
nmake -f makefile.vc install OPTS=static,nomsvcrt INSTALLDIR=
   path_to_your_install_dir TCLDIR=path_to_tcl_source
```

1.3.2 In Linux

Similarly you can build Tk in Linux using the following commands:

```
./configure --prefix=path_to_your_install_dir --enable-shared=no --with-tcl
   =path/to/tcl_source/unix
make
make install
```

1.4 Wrapping TCL Script

Lets assume now that you have the following script that you need to wrap with C in order to convert it to executable binary

```
puts "Hello World ......."
```

You can use the following primitive template in listing 1.1 to create a wrapper for your simple TCL script

Listing 1.1: Primitive Wrapper Template

```
#include <stdio.h>
#include <stdlib.h>
#include <tcl.h>

#define CMD_MAX_SIZE 256

static const char *user_script =
"puts \"Hello World .......\"\n"
;

int exe_tcl_cmd(Tcl_Interp *interp, const char *arg_str, ...){

  char *cmd_str = Tcl_Alloc(CMD_MAX_SIZE);
  va_list ap;
  int result=0;
  va_start(ap,arg_str);
  vsprintf(cmd_str, arg_str, ap);
  result = Tcl_Eval(interp,cmd_str);
  Tcl_Free(cmd_str);
  va_end(ap);

return result;
}
```

```
int main (int argc ,char *argv[]) {

  Tcl_FindExecutable(argv[0]);
  Tcl_Interp *myinterp;
  printf ("C: Starting ... \n");
  myinterp = Tcl_CreateInterp();

  if (Tcl_Init(myinterp) != TCL_OK) {
    printf("Error: %s\n",Tcl_GetStringResult(myinterp));
    return TCL_ERROR;
  }

  if (exe_tcl_cmd(myinterp, user_script) != TCL_OK) {
    printf("Error: %s\n",Tcl_GetStringResult(myinterp));
    return TCL_ERROR;
  }

  Tcl_DeleteInterp(myinterp);
  Tcl_Finalize();
  printf ("C: Finished\n");
  return TCL_OK;
}
```

Explaining the "TCL_..." functions is out of scope of this tutorial, but you can find some guidance in reference [7]. You can use this template for wrapping simple TCL scripts. Change the content of character pointer 'user_script' to contain your needed script but be careful about size of this 'user_script' as in this template we used 256 char. Remaining now to know how to compile this C wrapper to turn into executable file.

1.5 Compile C Wrapper for Windows

In this step we need again "Developer Command Prompt" of Visual Studio as we will use 'cl' command to create the executable. So navigate to directory of your C wrapper code to compile it. Building the wrapper differs in case you need to build your wrapper as standalone or to use TCL shared library.

1.5.1 Building with Shared Library

In case you can use TCL shared library, the following command can be used to compile the C wrapper

```
cl /c /I C:\path\to\TCL_Build\include /Tc user_code.c
cl user_code.obj /link C:\path\to\TCL_Build\lib\tcl87.lib
```

Note here that we have made the build in two steps just to make it simple to understand associated directives with 'cl' command. First step will generate 'user_code.obj'. '/Tc' is used to tell the compiler that it is C code file. '/I' is used to direct the compiler to consider generated "include" directory from the TCL build step. '/c' is used to compile the code only without linking. i.e. to generate 'user_code.obj' only. In second step we just linked the generated 'user_code.obj' with shared library

'tcl87.lib'. Note here that we wrote tcl87.lib because we built TCL of version 8.7.3a but it can be any version according to what you have used.

The generated exe file can be executed only if it is accompanied with tcl.dll file generated from TCL build. i.e both should reside in the same directory. For review of 'cl' command options, you can find at [5] and for syntax and linker options at [6].

1.5.2 Building with Static Library

In this case we have to use TCL static library. But it is not enough to use the TCL static library, you have also to direct the compiler to compile the user code with special TCL flag called "STATIC_BUILD".

```
cl /c /I C:\path\to\TCL_Build\include /D STATIC_BUILD /Tc user_code.c
cl user_code.obj user32.lib netapi32.lib /Link /LIBPATH:C:\path\to\
   TCL_Build\lib tcl87s.lib tclstub87.lib
```

Note here we had to use additional libraries: user32.lib and netapi32.lib which are general windows libraries. Moreover, we had to use TCL generated static library called tclstub87.lib which can be found in "lib" directory in compiled TCL directory. Again '87' in "tcl87s.lib" & "tclstub87.lib" denotes for TCL compiled version, but it can be any other version.

1.5.3 Adding Icon and Rules file

You may need to have your executable with your customized icon. If you have your .ico file ready, you can use it in Windows operating systems by first creating one rules file (.rc file) with the following lines

```
#define ICON_ID 101
ICON_ID  ICON  "user_icon.ico"
```

The index of ICON_ID is just random number to define priority of icon (it is used actually if you defined multiple icons for your executable). Then just compile this rules file using rules compiler. In Visual Studio, you can use "rc" command. In Linux you can use "windres" command. So building the executable file will become as follows. Note that "rc" command will produce ".res" file which we need to include during linking.

```
rc user_rules.rc
cl /c /I C:\path\to\TCL_Build\include /D STATIC_BUILD /Tc user_code.c
cl user_code.obj user_rules.res user32.lib netapi32.lib /Link /LIBPATH:C:\
   path\to\TCL_Build\lib tcl87s.lib tclstub87.lib
```

Beside definition of user icon, If you are going to have GUI using Tk libraries, you may need to include one file called "tk_base.rc". This file defines for Tk some icons and buttons to be used in the GUI. This file can be found in you Tk source directory after you build it (Tk_source >> win >> rc). So what you have to do now is it include this tk_base.rc file into your binary and adjust your own rules file to include it.

```
#define ICON_ID 101
ICON_ID  ICON  "user_icon.ico"
```

```
#if STATIC_BUILD
  #include "tk_base.rc"
#endif
```

Compiling your rules file has to be modified also to be

```
rc -DSTATIC_BUILD -DBASE_NO_TK_ICON -I"path/to/Tk/win/rc" user_rules.rc
```

What we have added here is two flags: "STATIC_BUILD" to direct rule compilers to include tk_base.rc and "BASE_NO_TK_ICON" to enable using of user icon because if it is not defined, the tk_base.rc will use its own default icon as the executable icon. Moreover, we told the rc compiler where to find tk_base.rc file using the -I directive. Note that this syntax for Visual Studio rc compiler the user_rules.rc must be at last of the command.

1.5.4 Creating Icon

You may need to add icon as shown previously but you don't know how to create this .ico file. For that purpose, you can use one free tool called ImageMagick. The important thing is .ico file is not like any normal graphic file. The .ico file is composed of several resolutions of one single image. Therefore, if you have a PNG image and you would like to use it as icon of your executable, then you may use script provided in listing 1.2 but first you have to download Imagemagick source. You can find portable release for windows at [13].

Listing 1.2: Icon Generation

```
target_file=user
magick_dir=/d/path/to/Imagemagick
rm -f ${target_file}_*

"$magick_dir"/magick.exe convert ${target_file}.png -compress none -resize
   96x96 -type truecolor -background white -colorspace RGB  -colors 256
   BMP3:${target_file}_14_512.bmp

"$magick_dir"/magick.exe convert -define ICO:bit-depth=24 ${target_file}
   _14_512.bmp -background white \
         \( -clone 0 -resize 16x16   -colorspace RGB -type truecolor -
            colors 16  \) \
         \( -clone 0 -resize 16x16   -colorspace RGB -type truecolor -
            colors 256  \) \
         \( -clone 0 -resize 32x32   -colorspace RGB -type truecolor -
            colors 16  \) \
         \( -clone 0 -resize 32x32   -colorspace RGB -type truecolor -
            colors 256  \) \
         \( -clone 0 -resize 32x32   -colorspace RGB -type truecolor -
            depth 24 \) \
         \( -clone 0 -resize 48x48   -colorspace RGB -type truecolor -
            colors 16  \) \
         \( -clone 0 -resize 48x48   -colorspace RGB -type truecolor -
            colors 256  \) \
```

```
          \( -clone 0 -resize 48x48   -colorspace RGB -type truecolor -
             depth 24 \) \
          \( -clone 0 -resize 128x128 -colorspace RGB -type truecolor -
             colors 16777216  \) \
          \( -clone 0 -resize 256x256 -colorspace RGB -type truecolor -
             colors 16777216  \) \
             -delete 0    ${target_file}_icon.ico

"$magick_dir"/magick.exe identify ${target_file}_icon.ico
```

In listing 1.2, first step was to convert your PNG image to BMP that is .ico file is better generated from BMP format. If your image already in BMP format, no need then for this conversion step.

Next, we generate the .ico file with different resolutions (16x16, 32x32, 48x48, 128x128, 256x256). You are free to produce more resolutions as you wish but note that width and height of the components are always the same. Each resolution also is produced with different colors. To control generated colors of the .ico component, you can either use "color" option or "depth" option. For example "-depth 24" and "-colors 16777216" are equivalent. This "colors" option describes how many bits are used to store each pixel. Also the "depth" means how many bits are used ($2^{24} = 16777216$). Therefore, they are equivalent. Last command "identify" is used to list details of the created .ico file.

1.6 Building Standalone TCL Application

Previous section showed how to generate executable that can run independent of any further system compiled libraries, however TCL has it own libraries that are used during run time. These libraries or packages or modules can be found in "lib" directory of the compiled TCL source. Therefore, if your application will need these packages/modules (definitely it will), these packages/modules have to be presented in specific location so that TCL can refer to it during run time. We couldn't compile these packages/modules in previous section within the tcl87s.lib as these packages/modules are pure TCL scripts and are not presented with a C API functions. Therefore, we need to find a way to include these TCL package/modules within our executable so that we can have totally standalone executable file.

1.6.1 Zipped Virtual File System

Fortunately there is one technique used for that purpose based on attaching TCL packages/modules to the executable file. This technique can be implemented using what is called zipped virtual file system. Initially there was one C code called zvfs.c that was used for that purpose which you can download from [9]. This code is quite old actually and is not compatible with recent TCL distributions. In addition, you need to compile zvfs.c and include it also into your binaries. However, ZVFS is still active in some applications like freewrap. But starting from TCL8.7, there is similar alternative for zvfs can be built by default in TCL core, this alternative is called ZipFS. You can use directly in your C wrapper. So, we can add couple of lines in our wrapper template to use the virtual file system as shown in listing 1.3

Listing 1.3: Wrapper Template with Zipfs

```
#include <stdio.h>
#include <stdlib.h>
#include <tcl.h>

#define STR_TCL_VERSION "tcl8.7"
#define virt_mount "/zipfs"
#define CMD_MAX_SIZE 256

static const char *user_script =
"source zipfs:/zipfs/user_code.tcl\n";

int exe_tcl_cmd(Tcl_Interp *interp, const char *arg_str, ...){

  char *cmd_str = Tcl_Alloc(CMD_MAX_SIZE);
  va_list ap;
  int result=0;
  va_start(ap,arg_str);
  vsprintf(cmd_str, arg_str, ap);
  result = Tcl_Eval(interp,cmd_str);
  Tcl_Free(cmd_str);
  va_end(ap);

return result;
}

int main (int argc ,char *argv[]) {

  Tcl_FindExecutable(argv[0]);
  Tcl_Interp *myinterp;
  printf ("C: Starting ... \n");
  myinterp = Tcl_CreateInterp();

    if (TclZipfs_Init(myinterp) != TCL_OK) {
        printf("Error in initializing Virtual File System\n");
        return TCL_ERROR;
    }
    if (TclZipfs_Mount(myinterp, virt_mount, Tcl_GetNameOfExecutable(),
       NULL) != TCL_OK) {
        printf("Error in mounting Virtual File System\n");
        return TCL_ERROR;
    }
    sprintf(tcl_dir, "zipfs:%s/%s", virt_mount, STR_TCL_VERSION);
    Tcl_SetVar2(myinterp, "env", "TCL_LIBRARY", tcl_dir, TCL_GLOBAL_ONLY);

  if (Tcl_Init(myinterp) != TCL_OK) {
    printf("Error: %s\n",Tcl_GetStringResult(myinterp));
    return TCL_ERROR;
```

```
  }

  if (exe_tcl_cmd(myinterp, user_script) != TCL_OK) {
    printf("Error: %s\n",Tcl_GetStringResult(myinterp));
    return TCL_ERROR;
  }

  Tcl_DeleteInterp(myinterp);
  Tcl_Finalize();
  printf ("C: Finished\n");
  return TCL_OK;
}
```

Note here that we used two additional functions “TclZipfs_Init” & “TclZipfs_Mount” to initialize the virtual file system and to mount it in TCL interpreter. Mounting here means that it will be seen only by the TCL interpreter for reading and execution only. i.e you can’t modify this virtual file system. For “TclZipfs_Mount” we need to mention where the virtual file system should be mounted, for instance in our template here we defined macro virt_mount as ”/zipfs”. This will be like home directory of your virtual system. Note also that we changed content of ’user_script’ to be:

```
static const char *user_script =
"source zipfs:/zipfs/user_code.tcl\n";
```

this is to make you able to run any tcl script you would like. Note here that we provide path to the user script as “zipfs:/zipfs/user_code.tcl” as we will add this script in the home directory of the virtual file system. We will see that in details in a while. By default, the Zipfs feature of TCL is defining root of the virtual file system as “zipfs:” [12] but it can be “//zipfs:” according to your operating system.

After initializing and mounting the virtual file system, we need to define path of this virtual file system in the TCL interpreter environment, therefore we have used code lines:

```
    sprintf(tcl_dir, "zipfs:%s/%s", virt_mount, STR_TCL_VERSION);
    Tcl_SetVar2(myinterp, "env", "TCL_LIBRARY", tcl_dir, TCL_GLOBAL_ONLY);
```

Here we defined “TCL_LIBRARY” to indicate the path of tcl8.7 packages/modules. Note again here that we used “zipfs:” as root of the virtual file system but it can be “//zipfs:”.

1.6.2 Appending ZVFS

Remaining now to know how to attach TCL packages/modules into our executable. This can be done during compiling the C wrapper. Its idea is very simple: just compress and concatenate it to the executable file. The listing 1.5 shows makefile that can be used in case of Windows for building the executable.

1.6.3 Loading Tk and Other Packages

You may need to call some TCL packages or modules in your TCL script. Beside attaching the whole source source of the package or module in the wrapper executable, we need to load it into

TCL interpreter. In normal TCL environment, you can try " `puts $auto_path` " auto_path is a predefined variable in any TCL interpreter which tells the TCL interpreter where to find source of packages or modules. This is what we need to do, we need to tell the Interpreter where to find needed packages or modules (which already exist in attached zipfs directory)

In case you use Tk interface, Tk is provided with C API, so we can load it within our C wrapper. Listing 1.4 shows complete wrapper that can be used to wrap a TCL script that uses Tk and other packages.

1.6.4 Command Line Only Application

In case you don't need Tk interface but you rather need to have executable that runs with user input arguments, you just need then to define the global variable "argv" of TCL interpreter with the argv passed from C main function. The following lines shows how to do that

```
    app_args = Tcl_Merge(argc-1, argv+1);
    Tcl_SetVar(myinterp, "argv", app_args, TCL_GLOBAL_ONLY);
```

It might be useful also to pass number of arguments "argc" and script name "argv0" to the TCL interpreter. Listing 1.4 shows how to apply that in lines between" `#ifdef CLI_ONLY_APP ....  #endif` ". Again "CLI_ONLY_APP" is a used defined macro to tell the compiler whether to compile these lines or not according to user application.

Listing 1.4: Complete Wrapper Template

```
#include <stdio.h>
#include <stdlib.h>
#include <tcl.h>

#define STR_TCL_VERSION "tcl8.7"
#define STR_TK_VERSION "tk8.7"
#define STR_PKG_CMDLINE "cmdline"
#define STR_PKG_STRUCT "struct"
#define virt_mount "/zipfs"
#define CMD_MAX_SIZE 256

static const char *user_script =
"source zipfs:/zipfs/user_code.tcl\n";

int exe_tcl_cmd(Tcl_Interp *interp, const char *arg_str, ...){

  char *cmd_str = Tcl_Alloc(CMD_MAX_SIZE);
  va_list ap;
  int result=0;
  va_start(ap,arg_str);
  vsprintf(cmd_str, arg_str, ap);
  result = Tcl_Eval(interp,cmd_str);
  Tcl_Free(cmd_str);
  va_end(ap);
```

```
return result;
}

int main (int argc ,char *argv[]) {

char tcl_dir[CMD_MAX_SIZE];
char tk_dir[CMD_MAX_SIZE];
char pkg_autopath[CMD_MAX_SIZE];

static char Wait_x[] =
     "bind . <Destroy> {+if {\"%W\"==\".\"} exit}\n"
     "while 1 {vwait forever}";

  Tcl_FindExecutable(argv[0]);
  Tcl_Interp *myinterp;
  printf ("C: Starting ... \n");
  myinterp = Tcl_CreateInterp();

#ifdef CLI_ONLY_APP
    app_args = Tcl_Merge(argc-1, argv+1);
    // the result string is dynamically allocated using Tcl_Alloc;
    // the caller must eventually release the space using Tcl_Free.
    Tcl_SetVar(myinterp, "argv", app_args, TCL_GLOBAL_ONLY);
    Tcl_Free(app_args);
    sprintf(scr_argc, "%d", argc-1);
    Tcl_SetVar(myinterp, "argc", scr_argc, TCL_GLOBAL_ONLY);
    Tcl_SetVar(myinterp, "argv0", argv[0], TCL_GLOBAL_ONLY);
    Tcl_SetVar(myinterp, "tcl_interactive", "0", TCL_GLOBAL_ONLY);
#endif

    if (TclZipfs_Init(myinterp) != TCL_OK) {
        printf("Error in initializing Virtual File System\n");
        return TCL_ERROR;
    }
    if (TclZipfs_Mount(myinterp, virt_mount, Tcl_GetNameOfExecutable(),
       NULL) != TCL_OK) {
        printf("Error in mounting Virtual File System\n");
        return TCL_ERROR;
    }
    sprintf(tcl_dir, "zipfs:%s/%s", virt_mount, STR_TCL_VERSION);
    Tcl_SetVar2(myinterp, "env", "TCL_LIBRARY", tcl_dir, TCL_GLOBAL_ONLY);
    Tcl_SetVar(myinterp,"tcl_library",tcl_dir,TCL_GLOBAL_ONLY);
    Tcl_SetVar(myinterp,"tcl_libPath",tcl_dir,TCL_GLOBAL_ONLY);
#ifdef ENABLE_TK
    sprintf(tk_dir, "zipfs:%s/%s", virt_mount, STR_TK_VERSION);
    Tcl_SetVar2(myinterp, "env", "TK_LIBRARY", tk_dir, TCL_GLOBAL_ONLY);
    Tcl_SetVar(myinterp, "tk_library", tk_dir, TCL_GLOBAL_ONLY);
#endif
```

```
  if (Tcl_Init(myinterp) != TCL_OK) {
    printf("Error: %s\n",Tcl_GetStringResult(myinterp));
    return TCL_ERROR;
  }
#ifdef ENABLE_TK
    if (Tk_Init(myinterp) != TCL_OK ){
        printf("Error: %s\n",Tcl_GetStringResult(myinterp));
        return TCL_ERROR;
    }
    Tcl_StaticPackage(myinterp,"Tk", Tk_Init, 0);
#endif

     // other user packages or modules
    sprintf(pkg_autopath, "zipfs:%s/%s", virt_mount, STR_PKG_CMDLINE);
    Tcl_SetVar(myinterp, "::auto_path", pkg_autopath, TCL_APPEND_VALUE |
       TCL_LIST_ELEMENT);
    sprintf(pkg_autopath, "zipfs:%s/%s", virt_mount, STR_PKG_STRUCT);
    Tcl_SetVar(myinterp, "::auto_path", pkg_autopath, TCL_APPEND_VALUE |
       TCL_LIST_ELEMENT);

  if (exe_tcl_cmd(myinterp, user_script) != TCL_OK) {
    printf("Error: %s\n",Tcl_GetStringResult(myinterp));
    return TCL_ERROR;
  }

#ifdef ENABLE_TK
    Tcl_Eval(myinterp,Wait_x);
#endif

  Tcl_DeleteInterp(myinterp);
  Tcl_Finalize();
  printf ("C: Finished\n");
  return TCL_OK;
}
```

So what we did here is initializing Tk package the same way we initialized TCL (that is because Tk is also provided with C APIs). For other packages (cmdline & struct in our example here) we used:

```
sprintf(pkg_autopath, "zipfs:%s/%s", virt_mount, STR_PKG_CMDLINE);
Tcl_SetVar(myinterp, "::auto_path", pkg_autopath, TCL_APPEND_VALUE |
   TCL_LIST_ELEMENT);
sprintf(pkg_autopath, "zipfs:%s/%s", virt_mount, STR_PKG_STRUCT);
Tcl_SetVar(myinterp, "::auto_path", pkg_autopath, TCL_APPEND_VALUE |
   TCL_LIST_ELEMENT);
```

Here we appended path of these packages into the global “auto_path” variable. It worth to mention here that these packages can be found or downloaded from one library called tcllib. At

time of writing this tutorial, tcllib-1.20 was available at link [14].

1.7 Compile C Wrapper for Linux

Listing 1.4 can be used in case we need to build an executable for Linux. No modification is needed except for the mounting driver. In Windows, we have used "zipfs:" but in Linux we have to use "//zipfs:". So any line that has "zipfs:" has to be changed to "//zipfs:". For example:

```
static const char *user_script =
"source //zipfs:/zipfs/user_code.tcl\n";
```

Better approach will be to define the drive name as a macro to be easy to change. But for our context here, we need to elaborate or explain how to build the wrapper so it was more convenient to simplify the explained C codes. If you had a look over tclpot application, you will find the C code is better constructed.

1.8 Compile Using Makefile

Since compiling the source code may require a lot of directives, it might be useful then to have a makefile that combines all user directives and options to ease the step of compiling the source files. We will go through makefiles examples for Windows and Linux operating systems.

1.8.1 Windows Makefile

Listing 1.5: WindowsMakfile

```
C_WRAPPER = user_code
TCL_SCRIPT = user_code
RULES_FILE = user_rules

TCL8_LIB_PATH="C:\path\to\TCL_Build\lib\tcl8"
TCL_LIB_PATH="C:\path\to\TCL_Build\lib\tcl8.7"
TK_LIB_PATH="C:\path\to\TCL_Build\lib\tk8.7"
TCL_PKG_CMDLINE="C:\path\to\tcllib-1.20\modules\cmdline"
TCL_PKG_STRUCT="C:\path\to\tcllib-1.20\modules\struct"

RULES_INCLUDE = -I"C:\path\to\tk8.7a3\win\rc"
INCLUDE_DIR = -I"C:\path\to\TCL_Build\include"
TCL_LIB_DIR = -LIBPATH:"C:\path\to\TCL_Build\lib"

COMPILER = cl
RULES_COMPILER = rc
CFLAGS = -DSTATIC_BUILD -DENABLE_TK -DBASE_NO_TK_ICON
COMPILE_FILES = /Tc $(C_WRAPPER).c
OBJECT_FILES = $(C_WRAPPER).obj $(RULES_FILE).res
SYSTEM_LIBS = user32.lib netapi32.lib kernel32.lib gdi32.lib
TCL_LIBS = tcl87s.lib tclstub87.lib
```

```
TK_LIBS = tk87s.lib tkstub87.lib

ZIP_TOOL = "C:\Program Files\7-zip\7z.exe"
ZIPPED_TCL_FILE=zipped_tcl_lib
UNZIPPED_TCL_LIB=TCL_RUNTIME_LIB


#--------------------------------------------------------------------
# Building
#--------------------------------------------------------------------
build:
    @rm -f $(ZIPPED_TCL_FILE).zip
    @rm -rf $(UNZIPPED_TCL_LIB)
    @mkdir $(UNZIPPED_TCL_LIB)
        @echo Building executable for user TCL script ....
    $(RULES_COMPILER) $(CFLAGS) $(RULES_INCLUDE) $(RULES_FILE).rc
    $(COMPILER) -c $(INCLUDE_DIR) $(COMPILE_FILES) $(CFLAGS)
    $(COMPILER) $(OBJECT_FILES) $(SYSTEM_LIBS) -link $(TCL_LIB_DIR) $(
       TCL_LIBS) $(TK_LIBS)
    @strip $(C_WRAPPER).exe
    @cp $(TCL_SCRIPT).tcl $(UNZIPPED_TCL_LIB)
    @cp -r $(TCL8_LIB_PATH) $(UNZIPPED_TCL_LIB)
    @cp -r $(TCL_LIB_PATH) $(UNZIPPED_TCL_LIB)
    @cp -r $(TK_LIB_PATH) $(UNZIPPED_TCL_LIB)
    @cp -r $(TCL_PKG_CMDLINE) $(UNZIPPED_TCL_LIB)
    @cp -r $(TCL_PKG_STRUCT) $(UNZIPPED_TCL_LIB)
    @cd $(UNZIPPED_TCL_LIB)
    $(ZIP_TOOL) a -r -tzip ..\$(ZIPPED_TCL_FILE).zip *
    @cd ..
    @cat $(ZIPPED_TCL_FILE).zip >> $(C_WRAPPER).exe
```

What we have done here is: firstly, compile user rules file with flags:

STATIC_BUILD to have a standalone executable. This is required by TCL C APIs.

ENABLE_TK which is a user defined macro that is used in listing 1.4.

BASE_NO_TK_ICON which is required if user wants to use his own customized ".ico" file as an icon for his executable.

The rc compiler needs to find also where to find tk_base.rc file. If we need to wrap TCL only script, no flags or include directive are needed for the rc compiler.

Afterwards, we can compile the C wrapper to generate exe file just as normal steps. The 'strip' command is optional to use. Afterwards, we copied user script, source of tcl8.7 libs,tk8.7 libs and user needed packages/modules to one new directory to be zipped together with user TCL script. Next is to zip these sources, Here we used 7-zip tool in windows to do the task. 7-zip tool has its own command format as we can see. i.e. we have to use "a -r -tzip" to add our sources into a zip file. After zipping the packages/libraries, we can use 'cat' command to append the zipped files into

our executable. If you created just a makefile in windows, you can run it in Visual Studio Command Prompt using nmake command.

```
nmake -f makefile.vc
```

It worth mentioning here that some Microsoft Visual Studio version is able to run Linux utilities (cp, cat, strip, ...). If this is not the case with your Microsoft Visual Studio, then you can use Windows utilities

```
RM    = del /q
RMDIR = rd /q /s
MKDIR = mkdir
ECHO  = echo
COPY  = copy
CPDIR = xcopy /q /i /e
CD    = cd
CAT   = type
```

Note that “strip” utility has no equivalent in Windows.

1.8.2 Linux Makefile

Similar Makefile can be used for Linux also, there will tiny changes.

Listing 1.6: Linux Makfile

```
C_WRAPPER = user_code
TCL_SCRIPT = user_code
RULES_FILE = user_rules

TCL8_LIB_PATH="/path/to/TCL_Build/lib/tcl8"
TCL_LIB_PATH="/path/to/TCL_Build/lib/tcl8.7"
TK_LIB_PATH="/path/to/TCL_Build/lib/tk8.7"
TCL_PKG_CMDLINE="/path/to/tcllib-1.20/modules/cmdline"
TCL_PKG_STRUCT="/path/to/tcllib-1.20/modules/struct"

RULES_INCLUDE = -I"/path/to/tk8.7a3/win/rc"
INCLUDE_DIR = -I"/path/to/TCL_Build/include"
TCL_LIB_DIR = "/path/to/TCL_Build/lib"

COMPILER = gcc
RULES_COMPILER = windres
CFLAGS = -DSTATIC_BUILD -DENABLE_TK -DBASE_NO_TK_ICON
COMPILE_FILES = $(C_WRAPPER).c
OBJECT_FILES = $(C_WRAPPER).o
SYSTEM_LIBS = -lz -ldl -lm -lpthread  -lX11 -lXft -lfontconfig -lXss
TCL_LIBS = $(TCL_LIB_DIR)/libtcl8.7.a $(TCL_LIB_DIR)/libtclstub8.7.a
TK_LIBS =  $(TCL_LIB_DIR)/libtk8.7.a  $(TCL_LIB_DIR)/libtkstub8.7.a

ZIP_TOOL = zip
```

```
ZIPPED_TCL_FILE=zipped_tcl_lib
UNZIPPED_TCL_LIB=TCL_RUNTIME_LIB


#--------------------------------------------------------------------
# Building
#--------------------------------------------------------------------
build:
    @rm -f $(ZIPPED_TCL_FILE).zip
    @rm -rf $(UNZIPPED_TCL_LIB)
    @mkdir $(UNZIPPED_TCL_LIB)
        @echo Building executable for user TCL script ....
    $(COMPILER) -c $(INCLUDE_DIR) $(COMPILE_FILES) $(CFLAGS)
    $(COMPILER) $(OBJECT_FILES) $(SYSTEM_LIBS) -link $(TCL_LIB_DIR) $(
       TCL_LIBS) $(TK_LIBS)
    @strip $(C_WRAPPER).exe
    @cp $(TCL_SCRIPT).tcl $(UNZIPPED_TCL_LIB)
    @cp -r $(TCL8_LIB_PATH) $(UNZIPPED_TCL_LIB)
    @cp -r $(TCL_LIB_PATH) $(UNZIPPED_TCL_LIB)
    @cp -r $(TK_LIB_PATH) $(UNZIPPED_TCL_LIB)
    @cp -r $(TCL_PKG_CMDLINE) $(UNZIPPED_TCL_LIB)
    @cp -r $(TCL_PKG_STRUCT) $(UNZIPPED_TCL_LIB)
    cd $(UNZIPPED_TCL_LIB); $(ZIP_TOOL) -r -q ..\$(ZIPPED_TCL_FILE).zip *;
       cd ..
    @cat $(ZIPPED_TCL_FILE).zip >> $(C_WRAPPER).exe
```

The first difference is Linux is using slashes "/" but Windows is using backslashes "\". C compiler in Linux will be gcc. Rules compiler (if used) will be windres. System libraries in Linux is different

```
SYSTEM_LIBS = -lz -ldl -lm -lpthread  -lX11 -lXft -lfontconfig -lXss
```

For zipping the executable, "zip" utility has been used. Note that inside the rule, the changing the directory and zipping the executable have to be performed in one line (not in separate lines like in Windows)

```
cd $(UNZIPPED_TCL_LIB); $(ZIP_TOOL) -r -q ..\$(ZIPPED_TCL_FILE).zip *; cd
   ..
```

This is very simple and primitive Makefile for explanation purpose. But for real efficient Makefile, it can be enhanced as implemented in tclpot application. Then finally of course, this Makefile is executed in Linux using make utility.

Chapter 2

Tclpot Guide

Tclpot is a free utility that have been created based on explanations of previous chapter and it is inspired by mktclapp and Tobe utilities. It can be obtained from (www.github.com/electgon/tclpot) for free under the GNU General Public License. It is provided 'as-is' and it is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Tclpot source is provided in tclpot.c. This C source file is explained in previous chapter. This chapter explains the build process which is better provided in a make file to conclude most needed build steps.

You can obtain tclpot by:

```
git clone https://github.com/electgon/tclpot
```

Before starting to wrap your TCL script make sure that you have compiled TCL source files available in your machine. If you are not sure how to compile TCL source files, please check www.electgon.com or check documentation distributed with Tclpot source repository. To wrap your TCL script, basically you need to issue this command In Windows:

```
nmake -f makefile.vc
```

In Linux:

```
make -f Makefile
```

But before that, open the make file and make sure that the following parameters are set correctly

C_WRAPPER	this is name of the tclpot.c file. If you renamed this c file, update this parameter accordingly.
TCL_SCRIPT	to define name of user tcl script that needs to be wrapped (just the name without .tcl extension).
RULES_FILE	if you have rules file (.rc) mention its name here (just the name without .rc extension).
PROJECT	if your script is based on TCL scripts only (no GUI), then make sure that this parameter is set to tcl. Otherwise, if your script is based on Tk, set this parameter to tk.

TCL8_NAME	It was noticed that compiling TCL source code in Windows results in separate directory called 'tcl8' which contains some old packages of TCL. In Linux, these packages are included in same main TCL packages directory. So, In Linux no need for this parameter but In Windows it will be needed and usually user doesn't need to change it.
TCL_VER_NAME	this should be name of the main directory that contains basic TCL packages. i.e. the directory that contains init.tcl file.
TK_VER_NAME	this should be name of the main directory that contains basic Tk packages. i.e. the directory that contains tk.tcl file.
<pkg>_PKG_NAME	If any other TCL package is needed within user TCL scrip, define a parameter with this needed package name.
PKGFLAGS	this parameter is used to pass previously mentioned package names to C source file which is using these names to define for the TCL interpreter where to find these used packages. If you didn't add your target package into this flag, then you have to provide packages names manually in the C source file.
TCL8_LIB_PATH	path to compiled tcl sources of version x which is applicable only in case of Windows as mentioned shortly.
TCL_LIB_PATH	path to compiled TCL sources of version x.x.
TK_LIB_PATH	path to compiled tk sources of version x.x.
TCL_PKG_<pkg>	if needed, provide path to user needed package <pkg> wich can be obtained from tcllib sources.
USE_RC_FILE	specify in this parameter whether you need to compile rc file or not.
RULES_INCLUDE	if you use Tk interface, you may need to include tk_base.rc file. mention here path to this file.
INCLUDE_DIR	TCL header files must be included during compile step. mention here path to include directory of TCL compiled source.
TCL_LIB_DIR	when you wrap your TCL with static libraries, mention here path to library directory that contains your TCL libs.
SYSTEM_LIBS	some general libraries of Windows or Linux are mentioned here. In principle no need to change this parameter.
TCL_LIBS	mention here name of TCL compiled libraries (shared or static). For example tcl87s.lib tclstub87.lib.
TK_LIBS	mention here name of Tk compiled libraries (shared or static). For example tk87s.lib tkstub87.lib.

CFLAGS	choose which macros are needed. Usually ”-DSTATIC_BUILD” will be needed if your TCL script is using Tk interface, use ”-DENABLE_TK” if your TCL script is tcl only with input arguments (no Tk), use ”-DCLI_ONLY_APP”. if you inlcuded tk_base.rc file and still you need to use your own icon, use ”-DBASE_NO_TK_ICON” ”PKGFLAGS” is ued to pass name of user TCL script (TCL_SCRIPT) as well packages names to the wrapper, so that you can avoid changing the wrapper source file.
COMPILE_FILES	if you have other C files that you need to compile, list here name of these files beside basic C_WRAPPER.
OBJECT_FILES	in most cases you will not need to modify this parameter.
ZIP_TOOL	define here path to your available zip program. For example ”C:\Program Files\7-zip\7z.exe”
RULES_COMPILER	if needed, define here rc compiler of your system.
COMPILER	define here C compiler of your system.

You can use

```
make clean
```

to clean up your build directory.

Repository Content:

- tclpot.c: main C code that is used to wrap user TCL script.
- makefile.vc : make file used to build the executable for Windows systems.
- Makefile : make file used to build the executable for Linux systems.
- user_rules : sample rules file in case of user is interested in adding icon to his executable.
- tcl_inside.ico : default icon file to be used if rc file is enabled.
- tk_hello.tcl : example for a TCL script with Tk interface.
- tclpot_guide.pdf : this file.
- README.md : quick guide for running the make files.

Bibliography

[1] https://www.tcl.tk/software/tcltk/download.html

[2] https://www.tcl.tk/doc/howto/compile.html

[3] https://github.com/tcltk

[4] https://visualstudio.microsoft.com/vs/community/

[5] https://docs.microsoft.com/en-us/cpp/build/reference/compiler-options-listed-by-category?view=vs-2019

[6] https://docs.microsoft.com/en-us/cpp/build/reference/compiler-command-line-syntax?view=vs-2019

[7] https://wiki.tcl-lang.org/page/How+to+embed+Tcl+in+C+applications

[8] https://www.tcl.tk/man/tcl8.7/TclLib/contents.htm

[9] http://www.hwaci.com/sw/tobe/zvfs.html

[10] https://www.hwaci.com/sw/mktclapp/mktclapp.html

[11] https://www.hwaci.com/sw/mktclapp/quickref.html

[12] https://www.tcl.tk/man/tcl8.7/TclLib/zipfs.htm

[13] https://imagemagick.org/script/download.php

[14] https://www.tcl.tk/software/tcllib/