Creating New Tcl/Tk Commands Using C

Shyam Pather

December 3, 1996

1 Introduction

Tcl provides and extensive set of general purpose built-in commands. However, some applications require adding new commands to those that Tcl provides. Very often, these new commands cannot be implemented in Tcl, because of speed requirements, and the need for access to system-level services. For this reason, the designers of Tcl provided an easy way to write new Tcl commands using C.

2 Packages

Ousterhout advises that new commands be implemented in a general way, so that they can be used in many different Tcl/Tk applications. To this end, he suggests grouping related commands into "packages" that can be distributed to various users. See Ousterhout, Chapter 31.

In order to use a package, one would typically create a new wish-like interpreter that contains all the standard Tcl/Tk commands as well as the new ones that comprise the package. This approach is used by many common extension packages such as **Tix** and **incr Tcl**.

In this tutorial, we will create a simple package, and a wish-like interpreter that uses it. Our package consists of two rather trivial commands: one that adds two integers, and another that gets the process ID of the current process.

3 Creating a New Package

A package typically consists of a C function that performs all package initialization tasks, and a series of C functions that implement the package commands.

The initialization for our package is done by the function *MyInit()*, declared in `myinit.h`, and implemented in `myinit.c`. This function simply calls *Tcl_CreateCommand()* to register each of our two new commands. Among other things, *Tcl_CreateCommand()* is passed the name that the command is to have in the Tcl interpreter, and the address of a C function that implements it.

The package commands are implemented in two functions, *AddCmd* and *PidCmd*. These functions can be found in the source file `mycmds.c`.

All C functions that implement a new Tcl command are required to have a standard list of arguments. The argument lists of *AddCmd()* and *PidCmd()* illustrate this. The first argument, *ClientData*, will be explained in a later tutorial. The other arguments consist of a pointer to the Tcl interpreter, the number of arguments passed to the Tcl command, and the values of those arguments. C functions that implement Tcl command should only return either *TCL_OK* or *TCL_ERROR*. Any additional information to be returned to Tcl should be placed in the result field of the interpreter (explained later).

The *AddCmd()* function implements a new tcl command called "add". This command simply adds two integer arguments. Before proceeding, the function first checks the arguments it was passed in Tcl. If it was not passed two arguments, it puts a message into the result field of the interpreter, and returns *TCL_ERROR*. Returning *TCL_ERROR* causes the Tcl exception handling code to be invoked. This causes the Tcl script that issued the "add" command to be halted, and the error message to be displayed.

If the command received the correct number of arguments, it simply adds them, and puts the sum into the interpreter result. It then returns *TCL_OK*, which causes Tcl to proceed normally.

In this example, we assumed the result field of the interpreter was pre-allocated to be big enough to hold the string we were putting into it. This is allowed when the strings in question are short. For longer strings, more caution is needed. See sections 30.3 and 30.4 in Ousterhout for other ways to write the interpreter result.

The "pid" command is implemented in the *PidCmd()* function. This function uses the *getpid()* system call to get the process ID. This is a good example of a system service that cannot be accessed directly from Tcl, therefore requiring the use of C.

4 Registering a New Package

In order to use a new package, it has to be initialized, and its commands registered in a Tcl interpreter. The program contained in the file `mywish.c` creates a wish-like interpreter and registers the new commands in them.

The *main()* function is simply a wrapper function that calls *Tk_Main()*. *Tk_Main()* performs all the tasks necessary to start wish, but never returns since it eventually enters the Tk event loop which terminates only when the application closes.

Tk_Main() is passed the address of an application-specific initialization procedure named *Tcl_AppInit()*. This procedure initializes all the packages that our wish-like application needs. It first initializes all the standard Tcl commands by calling *Tcl_Init()*, then all the standard Tk commands by calling *Tk_Init()*, and then our new package by calling *MyInit()* (described in the previous section). Leaving out the call to *MyInit()* would result in an exact replica of wish.

5 Compiling and Linking

The example Makefile provided illustrates how to compile and link the new Tcl package and the new wish-like interpreter. In this case, the package initialization and implementation

code are compiled as separate modules, which are later linked into the executable file for the wish-like interpreter.

In the compile step for `mywish.c`, two symbolic constants are defined, namely *TCL_LIBRARY* and *TK_LIBRARY*. The values of these constants are the paths to directories where certain Tcl and Tk startup scripts are stored. Tcl and Tk use these values when constructing the paths to these startup scripts.

Also, note the libraries against which the final executable is linked. These include the tcl, tk, X11, math and C libraries, which are assumed to exist in the default locations.

6 Using the new commands

Successfully compiling the code will result in a new executable file called `mywish`. This is the new wish-like interpreter, which is invoked by typing `mywish` at the command prompt. When the wish-prompt appears, type `info commands` to get a list of commands that the interpreter understands. Somewhere in that list will be the two commands we just created, namely "pid" and "add". You will also notice that the new interpreter behaves exactly like wish, except that it knows about the two new commands.

7 Things to Try

Experiment with these new commands. Try giving the wrong number of arguments to the "add" command to see what happens in the case of an error.