## NAME

TclCommandWriting - Writing C language extensions to Tcl.

## OVERVIEW

This document is intended to help the programmer who wishes to extend Tcl with C language routines. It should also be useful to someone wishing to add Tcl to an existing editor, comm program, etc. There is also programming information in the *Tcl.man* manual directory of the Berkeley distribution.

## WRITING TCL EXTENSIONS IN C

C extensions to Tcl must be written to receive their arguments in the manner Tcl uses to pass them.

A C routine is called from Tcl with four arguments, a client data pointer, an interpreter pointer, an argument count and a pointer to an array of pointers to character strings containing the Tcl arguments to the routine.

A Tcl extension in C is now presented, and described below:

```
#include "tcl.h"

int Tcl_EchoCmd(clientData, interp, argc, argv)
    void       *clientData;
    Tcl_Interp *interp;
    int         argc;
    char      **argv;
{
        int i;

        for (i = 1; i < argc; i++) {
                printf("%s ",argv[i]);
        }
        printf("\n");
        return TCL_OK;
}
```

The client data pointer will be described later.

The interpreter pointer is the ‘‘key’’ to an interpreter. It is returned by *Tcl_CreateInterp* or *Tcl_CreateExtendedInterp* and is used within Tcl and by your C code. The structure pointed to by the interpreter pointer, and all of the subordinate structures that branch off of it, make up an interpreter context, which includes all of the currently defined procedures, commands, variables, arrays and the execution state of that interpreter.

The argument count and pointer to an array of pointers to textual arguments is handled by your C code in the same manner that you would use in writing a C *main* function -- the argument count and array of pointers works the same as in a C *main* call; pointers to the arguments to the function are contained in the *argv* array. Similar to a C main, the first argument (*argv[0]*) is the name the routine was called as (in a main, the name the program was invoked as).

In the above example, all of the arguments are output with a space between each one by looping through *argv* from one to the argument count, *argc*, and a newline terminates the line -- an echo command, although a ‘‘real’’ echo command would not add a trailing blank like this one does.

All arguments from a Tcl call to a Tcl C extension are passed as strings. If your C routine expects certain numeric arguments, your routine must first convert them using the *Tcl_GetInt* or *Tcl_GetDouble* function, or some other method of your own devising. If you program produces a numeric result, it should return a string equivalent to that numeric value. A common way of doing this is something like...

```
sprintf(interp->result, "%ld", result);
```

More sophisticated commands should verify their arguments when possible, both by examining the argument count, by verifying that numeric fields are really numeric, that values are in range when their ranges are known, and so forth.

Tcl is designed to be as bullet-proof as possible, in the sense that Tcl programs should not be able to cause Tcl to dump core. Please do the same with your C extensions by validating arguments as above.

In the command below, two or more arguments are compared and the one with the maximum value is returned, if all goes well. It is an error if there are fewer than three arguments (the pointer to the ‘‘max’’ command text itself, *argv[0]*, and pointers to at least two arguments to compare the values of).

This routine also shows the use of the programmer labor-saving *Tcl_AppendResult* routine. See the Tcl manual page, *SetResult.man*, for details. Also examine the calls *Tcl_SetErrorCode* and *Tcl_UnixError* documented in the Tcl manual page *AddErrInfo.man*.

```
int
Tcl_MaxCmd (clientData, interp, argc, argv)
    char       *clientData;
    Tcl_Interp *interp;
    int         argc;
    char      **argv;
{
    int maxVal = MININT;
    int maxIdx = 1;
    int value, idx;


    if (argc < 3) {
        Tcl_AppendResult (interp, "bad # arg: ", argv[0],
                          " num1 num2 [..numN]", (char *)NULL);
        return TCL_ERROR;
    }

    for (idx = 1; idx < argc; idx++) {
        if (Tcl_GetInt (argv[idx], 10, &Value) != TCL_OK)
            return TCL_ERROR;

        if (value > maxVal) {
            maxVal = value;
            maxIdx = idx;
        }
    }
    strcpy (interp->result, argv [maxIdx]);
    return TCL_OK;
}
```

When Tcl-callable functions complete, they should normally return **TCL_OK** or **TCL_ERROR**. **TCL_OK** is returned when the command succeeded and **TCL_ERROR** is returned when the command has failed rather drastically. **TCL_ERROR** should be returned for all syntax errors, non-numeric values where numeric ones were expected, and so forth. Less clear in some cases is whether Tcl errors should be returned or whether a function should just return a status value. For example, end-of-file during a *gets* returns a status, but *open* returns an error if the open fails. Errors can be caught from Tcl programs using the *catch* command.

Less common return values are **TCL_RETURN**, **TCL_BREAK** and **TCL_CONTINUE**. These are used if you are adding new control and/or looping structures to Tcl. To see these values in action, examine the source to the *while*, *for*, *if* and *loop* commands.

## INSTALLING YOUR COMMAND

To install your command into Tcl your must call *Tcl_CreateCommand*, passing it the pointer into the interpreter you want to install the command into, the name of the command, a pointer to the C function, a client

data pointer, and a pointer to an optional callback routine.

The client data pointer and the callback routine will be described later.

For example, for the max function above (which incidentally comes from math.c in the extend/src directory):

```
Tcl_CreateCommand (interp, "max", Tcl_MaxCmd, (ClientData)NULL,
                   (void (*)())NULL);
```

In the above example, the max function is added to the specified interpreter. The client data pointer and callback function pointer are NULL.

## CLIENT DATA

The client data pointer provides a means for Tcl commands to have data associated through them that is not global to the C program including the Tcl core. It is essential in a multi-interpreter environment (where a single program has created and is making use of multiple Tcl interpreters) for the C routines to maintain any permanent data they need relative to each interpreter being used, or there would be reentrancy problems. Tcl solves this through the client data mechanism. When you are about to call *Tcl_CreateCommand* to add a new command to an interpreter, if that command needs to keep some read/write data from one invocation to another, you should allocate the space, preferably using *ckalloc*, then pass the address of that space as the ClientData pointer to *Tcl_CreateCommand*.

When your command is called from Tcl, the ClientData pointer you gave to *Tcl_CreateCommand* when you added the command to that interpreter is passed to your C routine through the ClientData pointer calling argument.

Commands that need to share this data with one another can do so by using the same ClientData pointer when the commands are added.

It is important to note that the Tcl extensions in the extended/src directory have had all of their data set up in this way, so at the time of this writing (release 6.2) the Tcl extensions support multiple interpreters within one invocation of Tcl.

## INTEL ’286 GOTCHAS

The ’286 programmer who is not using an ANSI C standard compiler with function prototypes must be vigilant to ensure that anytime NULL is passed to a function as a pointer it is explicitly cast to **(void *)** or equivalent. Also remember that Tcl math within expressions is carried out to 32 bits, so that you should usually use the *long* variable type for your integers, *Tcl_GetLong* (rather than *Tcl_GetInt*) to convert strings to long integers, and remember to use **%ld** when printing results with *sprintf*, and so forth.

To maintain ’286 compatibility, all C programmers are asked to follow these guidelines. I know you don’t want to, but there are a lot of 286 machines out there and it is nice that they are able to run Tcl.

## THEORY OF HANDLES

Sometimes you need to have a data element that isn’t readily representable as a string within Tcl, for example a pointer to a complex C data structure. We do not think it is a good idea to try to pass pointers around within Tcl as strings by converting them to and from hex or integer representations, for example. It is too easy to screw one up and the likely outcome of doing that is a core dump.

Instead what we have done is developed and made use of the concept of *handles*. Handles are identifiers a C extension can pass to, and accept from, Tcl to make the transition between what your C code knows something as and what name Tcl knows it by to be as safe and painless as possible. For example, the stdio package included in Tcl uses file handles. When you open a file from Tcl, a handle is returned of the form **file***n* where *n* is a file number. When you pass the file handle back to *puts*, *gets*, *seek*, *flush* and so forth, they validate the file handle by checking the the **file** text is present, then converting the file number to an integer that they use to look into a data structure of pointers to Tcl open file structures, which contain a Unix file descriptor, flags indicating whether or not the file is currently open, whether the file is a file or a pipe and so forth.

Handles have proven so useful that, as of release 6.1a, general support has been added for them. If you

need a similar capability, it would be best to use the handle routines, documented in *Handles.man*. We recommend that you use a unique-to-your-package textual handle coupled with a specific identifier and let the handle management routines validate it when it's passed back. It is much easier to track down a bug with an implicated handle named something like **file4** or **bitmap6** than just **6**.

## TRACKING MEMORY CORRUPTION PROBLEMS

Occasionally you may write code that scribbles past the end of an allocated piece of memory. The memory debugging routines included in Tcl can help find these problems. See *Memory(TCL)* for details.

## WRITING AN APPLICATION-SPECIFIC MAIN

For those writing an application-specific main, for example, those adding Tcl to an existing application or including Tcl within a larger application, a few steps need to be taken to set up Tcl.

For one thing, several *extern char* * definitions must be fulfilled, providing data used by the *infox* command. These definitions are *tclxVersion*, the Extended Tcl version number, *tclxPatchlevel*, the Extended Tcl patch level, *tclAppName*, the name of the application, *tclAppLongname*, a description of the application, and *tclAppVersion*, the version number of the application.

A Tcl interpreter, including all of the extensions in Extended Tcl, is created with a call to *Tcl_CreateExtendedInterp*. Next, any application-specific commands are added by calls to *Tcl_CreateCommand*. Finally, *Tcl_Startup* is called to load the Tcl startup code, pull in all of the Tcl procs and paths, do command line processing, handle autoloads, packages, and so forth. If the application writer wants different startup behavior, they should write a different Tcl startup routine. *Tcl_Startup* is defined in the file *tclstartup.c* in the *extended/src* directory.

Finally, cleanup code is called to close down the application. *Tcl_DeleteInterp* is called to free memory used by Tcl -- normally, this is only called if **TCL_MEM_DEBUG** was defined, since Unix will return all of the allocated memory back to the system, anyway.

The writer of an application-specific main is invited to examine and use the *main()* routine defined in *extended/src/main.c* as a template for their new main. There is a *tcl++.C*, which is a main for C++-based Tcl applications.