TASH: Tcl Ada SHell, An Ada/Tcl Binding

©1996, Terry J. Westley
Calspan SRL Corporation
twestley@acm.org

Abstract

A binding to Tcl from Ada is described. The goal of this binding is to make scripting language features, such as associative arrays, regular expression matching, and execution of OS commands available to an Ada programmer and to allow a Tcl programmer to use Ada in place of C where needed. This binding exploits several new features of Ada 95 that make interfacing to C much cleaner and more direct than Ada 83.

The Ada Programming Language

The Ada programming language is a wonderful integration of traditional mainstream programming language design and newer software engineering features proven in more recent languages. The result is a language with great expressive power, but also with safety and reliability. New features added by Ada 95 promise further improvements.

The Tcl Programming Language

Tcl (Tool Command Language) is a simple, but powerful, scripting language invented by John Ousterhout at University of California at Berkeley[1]. It is so very different from Ada: interpreted, no data typing, poor management of the name space, and limited modularity. Obviously, this is because Tcl was designed for a different purpose than was Ada. But, Dr. Ousterhout designed Tcl with excellent facilities for using C where appropriate. The resulting environment combines the advantages of a simple scripting language with the power of a compiled systems language.

Programming Languages

When starting a programming task, it is important to choose the right tools, especially the programming language. Every language has its strengths and weaknesses and these must be considered (among many other criteria) when making the choice. After choosing a language for a particular task, one often finds the need for features from another language. For example, if a Tcl program grows from a small script into a major program, the lack of good name space management and complex data structures (such as arrays of records) becomes a significant burden. In a similar way, an Ada programmer may find uses for associative arrays, regular expression matching and execution of OS commands found in scripting languages such as Tcl and Perl [5].

Ada/Tcl

With its embeddable library of functions, Tcl offers the opportunity to ameliorate this dilemma, as long as one uses C as the “host” language. But, is it possible to use Ada in place of C? With the advent of Ada 95, the task of writing a binding to C has proven to be not only possible, but also very clean and direct.

The major objectives of this effort have been to produce a binding which would:

- Allow a Tcl program to use Ada in place of C to implement Tcl commands where additional execution speed, more complex data structures, or better name space management was needed, and
- Make the Tcl library functions available to an Ada program. These include string and list handling, regular expression matching, hash tables, and access to the X window system (via Tk).

Introduction to Tcl

Tcl is:

Free: the source code is copyrighted by the Regents of the University of California and Sun Microsystems and is freely available (via anonymous ftp) with few restrictions,

Interpreted: although compilers are available, the code is more commonly interpreted,

General purpose: provides general programming facilities such as variables, iteration, condition evaluation, and procedures,

Extensible: the language can be easily extended in Tcl as well as in C. One of the most popular

---

[1] Dr. Ousterhout is now at Sun Microsystems, heading up a team which is developing PC and Mac versions of Tcl/Tk as well as commercial tools for supporting application development.

extensions is Tk, a toolkit for building graphical user interfaces based on the X Window System,

Embeddable: its interpreter is implemented as a library of C functions which can be incorporated into and extended by a larger application.

Portable: beginning with Tcl7.5 and Tk4.1, the core Tcl/Tk system is available for Macs and PCs as well as for Unix.

Some of the advantages of using Tcl in an application are:

Rapid development: Because of its extensibility and availability of add-on packages, Tcl, along with Tk, provide a higher level language than C. Being an interpreted language also results in quicker turn-around than the traditional edit-compile-link-test cycle of software development.

System integration: Tcl makes an excellent language for integrating several programs into one single user environment. Because it is embeddable, Tcl can also serve as an inter-application language. Many also hope it to be used in the future as an internetworking software agent language.

User programming: Where appropriate, users can program an application directly in the scripting language. Many very successful applications, such as AutoCad and Excel, provide user-level programming through a custom language. With Tcl, the application designer can provide for user programming without having to design and implement a new language.

Sample Tcl script

Figure 1 is a sample Tcl script. It reads standard input and produces a frequency count, treating each input line as a separate item. To count the frequency of each word in a document in a Unix environment, enter this command:

```
deroff -w file | freq
```

The **`deroff`** command replaces all word separators in a file with line terminators. Its output is piped to the input of the **`freq`** script. Word separation is not done in the **`freq`** script to keep the script pure in its purpose and implementation.

```
#!/usr/local/bin/tclsh

# read lines from standard input until
# end of file encountered
while {[gets stdin line] >= 0} {
   if [info exists Freq($line)] {
      # the item is already in the
array,
      # so just increment its count
      incr Freq($line)
   } else {
      # the item is not in the array
yet,
      # so initialize its count
      set Freq($line) 1
   }
}

# iterate through every item and print
it
# and its frequency count
foreach item [array names Freq] {
   puts stdout "$item $Freq($item)"
}
```

Figure 1

Notice the strong influence of C and scripting languages, such as the Unix C shell, in the syntax of Tcl. This program would not be very difficult to program in Ada, although coding the associative array Freq would prove to be somewhat verbose and time-consuming.

Introduction to the Ada/Tcl binding

The binding presented in this paper is an Ada binding to Tcl version 7.5a2. It is our intention to add a Tk binding in the future. Other plans include investigating the addition of other extensions, such as TclX, and a general extension facility.

TASH is the name of the interactive shell provided by this binding. It operates exactly as does the **`tclsh`** program distributed with Tcl. The name TASH is also used in this paper to refer to the binding itself.

"Thin" binding

This binding attempts to provide a faithful interface to the Tcl C library as represented in the Tcl header file, "**`tcl.h`**." One significant difference is that many Tcl data structures are declared **`private`** in the Ada version where they are not in C, although this appears to be the intention of the original design.

Figure 2 is an example of this style. It shows the interface, via pragma "import," for creating a Tcl interpreter.

```
type Interp_Rec is private;
type Interp_Ptr is access all
Interp_Rec;
function CreateInterp return Interp_Ptr;
pragma Import (C, CreateInterp,
   "Tcl_CreateInterp");
```

Figure 2

Every Tcl function is implemented with a **`pragma Import`** interface. This will be referred to as the "thin" binding because it uses the C data types in all function arguments and return values.

"Thick" binding

The CreateInterp function of Figure 2 is used in an Ada program as follows:

```
declare
   Interp : Tcl.Interp_Ptr;
begin
   Interp := Tcl.CreateInterp;
   ...
end;
```

This function may be used easily in conventional Ada programming styles. Use of other functions, such as Tcl.StringMatch, shown in Figure 3, might be considered awkward by an Ada programmer[2].

```
function StringMatch (
   Str     : in C.Strings.Chars_Ptr;
   Pattern : in C.Strings.Chars_Ptr)
      return C.Int;
pragma Import (C, StringMatch,
   "Tcl_StringMatch");
-- Returns 1 if Str matches Pattern
using
-- glob-style rules for pattern
matching,
-- 0 otherwise.
```

Figure 3

This is because it requires the use of integer result codes in place of Booleans and exception handlers and does not work conveniently with the standard Ada string type. This is demonstrated in the following code which uses the StringMatch function to check whether the string "tartar sauce" contains the pattern "tar:"

```
declare
   Result_Code : C.Int;
begin
   Result_Code := Tcl.StringMatch (
      Str => New_String ("tartar
sauce"),
      Pattern => New_String ("tar"));
end;
```

To our dismay, however, we have a created a problem greater than writing Ada in a verbose C style: memory leakage. Without automatic garbage collection (rare in Ada environments), the code above may consume memory. One solution is to declare the strings as accessible variables and free them explicitly:

```
declare
   Str : Chars_Ptr := New_String (
      "tartar sauce");
   Pat : Chars_Ptr := New_String
("tar");
   Result_Code : C.Int;
begin
   Result_Code := Tcl.StringMatch (
      Str, Pat);
   Free (Str);
   Free (Pat);
end;
```

Any good programmer is far too lazy to write code in this verbose manner! Our Ada/Tcl binding, through the "magic" of overloading, provides a reasonable set of additional subprograms for many of the primitive operations. Figure 4 shows those of StringMatch.

```
function StringMatch (
   Str     : in C.Strings.Chars_Ptr;
   Pattern : in C.Strings.Chars_Ptr)
      return Boolean;
function StringMatch (
   Str     : in String;
   Pattern : in C.Strings.Chars_Ptr)
      return Boolean;
function StringMatch (
   Str     : in C.Strings.Chars_Ptr;
   Pattern : in String) return Boolean;
function StringMatch (
   Str     : in String;
   Pattern : in String) return Boolean;
-- Returns True if Str matches Pattern
-- using glob-style rules for pattern
-- matching, False otherwise.
```

Figure 4

The purpose of the additional subprograms is to provide an interface which better reflects typical Ada usage. This includes the use of exceptions, standard Ada data types, procedure subprograms, default parameter values, and appropriate return types.

For small strings, such as variable names and values, the inefficiency of converting to a C string, then releasing its space, is minimal. Where it is not, the programmer has the choice to work in C strings directly. The combinations in input parameter types

[2]Throughout, assume Ada code examples are in the context of: "with Interfaces.C" and "package C renames Interfaces.C."

of standard Ada strings and C strings allows the programmer freedom to choose what is needed for each situation. Now, the search for "tar" can be coded as follows:

```
if Tcl.StringMatch (
      "tartar sauce", "tar") then
   Text_IO.Put_Line ("tar found");
end if;
```

More Thoughts on the "Thin" vs. "Thick" Binding

While building this interface, the author struggled for quite some time between a pure C interface (often called a "thin" binding) and a pure Ada interface ("thick" binding).

The first version was a simple "thin" binding. The resulting programming style was particularly unsatisfying as is demonstrated in the pervious section. More frustrating was the unavailability of the string handling functions for "native" strings in package Ada.Strings. The Interfaces.C package does not provide a similar set of functions for handling C strings.

The second version was a pure Ada interface, or "thick" binding. All the pragmas to C were hidden in the body of the package. The resulting programming style was more pleasing and convenient. However, the requirement to provide every feature of Tcl in a pure Ada binding proved very daunting with little apparent payback. For example, many Tcl capabilities are implemented with callbacks. It is essential that the subprogram called from Tcl have the correct specification, including the use of C data types.

At this point, other bindings, such as the POSIX Ada binding and several Ada bindings to the X Window System were reviewed for ideas. None of these offered a satisfactory solution for Tcl.

Finally, a compromise was reached in which both the "thin" and "thick" binding facilities were mixed in one package. Splitting these into two separate packages is feasible and may be desirable from a binding maintenance viewpoint. However, this makes the programmer's task of finding, selecting, and qualifying the needed features more difficult. Also, some of the overloaded subprograms provide mixtures of Ada and C data types. It is not clear whether these should go into the "thin" or "thick" layer as they have been defined here.

The string handling problem was resolved by writing a string handling package for the Interfaces.C.Strings.Chars_Ptr data type in the pattern of Ada.Strings.Unbounded. Although this package is not directly used in the Ada/Tcl binding, it is included with the distribution.

Naming conventions

All names in the C interface which begin with "Tcl_" were changed to remove this prefix. This was done with the intention that the Ada programmer would use fully qualified names, thus reducing the redundancy of a call such as "Tcl.Tcl_CreateInterp" to simply "Tcl.CreateInterp."

Although case is not significant in Ada, all identifiers are capitalized as in the C library to aid in reading and recognizing corresponding identifiers.

In cases where a C name is an Ada reserved word, the name was generally shortened, e.g. TCL_RETURN became Tcl.RETRN.

Choosing subprogram variants

As mentioned before, many variants of the subprograms were added to make the binding more directly usable in an Ada program. These guidelines were used in deciding where this was appropriate:

1) Add procedures where the original subprogram is a function and the return value is actually a completion code. This corresponds to common usage in C where a function is used as a procedure. In Ada, we can raise an exception where an error occurs. All such procedures in this binding raise the Tcl_Error exception when an error return code is returned from the Tcl function.

2) Replace C.Strings.Chars_Ptr input parameters with standard Ada string type. Where reasonable, also provide combinations of Chars_Ptr and String. The implementation of such subprograms takes care of converting String to Chars_Ptr and freeing them after use to prevent memory leaks.

3) Replace C.Strings.Chars_Ptr in function return types with standard Ada string type and handle all necessary conversions and memory management.

4) Replace use of pointers in C functions with "out", "in out", or "access" mode parameters where appropriate.

5) Replace C.Int in function return types with Boolean where appropriate.

6) **Don't** replace C.Int in function return types where an integer return value is appropriate. This minimizes the combinations of subprogram

variants without great sacrifice of speed or readability in the caller.

Client Data

The Tcl library uses a common technique for passing private types to C functions:

```
typedef void *ClientData;
```

Unfortunately, the compiler can provide no assistance in verifying consistent data type usage across function calls which should operate on the same data type.

This Ada/Tcl binding takes advantage of Ada generics to provide type consistency. Use of a tagged type with a common ancestor was rejected as it seemed unnecessary and would have required coupling of potentially very different data types.

To reduce the number of necessary instantiations, related functions using the same data type are collected in packages. The "man" pages in the Tcl distribution were used as guidelines to determine how to classify the ClientData functions. Figures 5 and 6 show an example of this: a subset of the C function prototypes (macros are incomplete) and Ada generic specification for hash table handling.

```
EXTERN void Tcl_InitHashTable
_ANSI_ARGS_((
   Tcl_HashTable *tablePtr, int
keyType));

EXTERN void Tcl_DeleteHashTable
_ANSI_ARGS_
   ((Tcl_HashTable *tablePtr));

#define Tcl_CreateHashEntry(\
   tablePtr, key, newPtr)

#define Tcl_GetHashValue(h)

#define Tcl_SetHashValue(h, value)
```

Figure 5

```
procedure InitHashTable (
   tablePtr : in HashTable_Ptr;
   keyType  : in C.Int);
pragma Import (C, InitHashTable,
   "Tcl_InitHashTable");

procedure DeleteHashTable (
   tablePtr : in HashTable_Ptr);
pragma Import (C, DeleteHashTable,
   "Tcl_DeleteHashTable");

function CreateHashEntry (
   tablePtr : in HashTable_Ptr;
   Key      : in C.Strings.Chars_Ptr;
   NewPtr   : in C_Aux.Int_Ptr) return
      HashEntry_Ptr;
pragma Inline (CreateHashEntry);

function GetHashValue (
   EntryPtr : in HashEntry_Ptr) return
      ClientData_Ptr;
pragma Inline (GetHashValue);

procedure SetHashValue (
   EntryPtr : in HashEntry_Ptr;
   Value    : in ClientData_Ptr);
pragma Inline (SetHashValue);
```

Figure 6

Variable-length Argument Lists

Variable-length argument lists are used in the Tcl library primarily to pass one or more strings to be made into a proper list or to construct a larger string. Since all Tcl variables are stored as strings, it is not necessary to pass lists of mixed data types as is required to implement an interface to "printf."

Thus, it was not considered necessary to implement a general-purpose interface to C "varargs" as described in [1] in order to pass arguments of different data types. This capability is being considered as a future addition, but for now, it appears to be adequate to simply provide a function interface which may take many arguments as shown in Figure 7. The body of these subprograms adds a tenth string argument, guaranteed to be a CS.Null_Ptr before calling the C function. This allows the programmer to utilize all 9 of the String arguments.

```
package CS renames Interfaces.C.Strings;

procedure AppendResult (
 interp  : in Interp_Ptr;
 String1 : in CS.Chars_Ptr;
 String2 : in CS.Chars_Ptr :=
CS.Null_Ptr;
 String3 : in CS.Chars_Ptr :=
CS.Null_Ptr;
 String4 : in CS.Chars_Ptr :=
CS.Null_Ptr;
 String5 : in CS.Chars_Ptr :=
CS.Null_Ptr;
 String6 : in CS.Chars_Ptr :=
CS.Null_Ptr;
 String7 : in CS.Chars_Ptr :=
CS.Null_Ptr;
 String8 : in CS.Chars_Ptr :=
CS.Null_Ptr;
 String9 : in CS.Chars_Ptr :=
CS.Null_Ptr);
```

Figure 7

Accessibility checks

In developing the TASH main program in the same style as tclAppInit.c, accessibility check errors were encountered using the Access attribute to reference both variables and subprograms. Figure 8 shows the C version of the Tcl shell main program.

```
int Tcl_AppInit(Tcl_Interp *interp);

int main(argc, argv)
   int argc;
   char **argv;
{
   Tcl_Main(argc, argv, Tcl_AppInit);
   return 0;
}
```

Figure 8

When attempting to duplicate the C model in Ada, the code in Figure 9 was produced. However, both uses of the Access attribute in the call to Tcl.Main cause accessibility errors because both Argv and AppInit are declared at a deeper accessibility level than are C.Int and Tcl.AppInitProc_Ptr.

```
procedure tash is

   Argc : C.Int := C.Int (
      Ada.Command_Line.Argument_Count) +
1;
   Argv : C_Aux.Arg_Vector(1..Argc);

   function AppInit (
         Interp : in Tcl.Interp_Ptr)
      return C.Int;

begin -- tash
   C_Aux.Get_Argv (Argv);
   Tcl.Main (Argc,
      Argv(Argv'first)'access,
      AppInit'access);
end tash;
```

Figure 9

Knowing that Argv will not go out of scope before completion of its use in Tcl.Main allows us to change the Access attribute referencing it to Unchecked_Access. Unfortunately, no equivalent to Unchecked_Access is available for subprogram references. Since Tcl.AppInitProc_Ptr is declared at library level, so must AppInit. Thus, the package TashApp was created to declare AppInit and so this package and the main program, TASH, together make up the Ada version of tclAppInit.c. This is shown in Figure 10.

Callbacks

Tcl uses callbacks extensively. These are defined as pointer to function typedefs in the C Tcl public interface (**`tcl.h`**), for example:

```
typedef int (Tcl_AppInitProc)
   _ANSI_ARGS_((Tcl_Interp *interp));
```

This example defines the interface to a function, to be provided by the user, which performs the application-specific initialization.

```
package TashApp is
   function Init (
         Interp : in Tcl.Interp_Ptr)
      return C.Int;
   pragma Convention (C, Init);
end TashApp;

-----------------------------------------
-

with TashApp;

procedure tash is

   Argc : C.Int := C.Int (
      Ada.Command_Line.Argument_Count) +
1;
   Argv : C_Aux.Arg_Vector(1..Argc);

begin -- tash
   C_Aux.Get_Argv (Argv);
   Tcl.Main (Argc,
      Argv(Argv'first)'unchecked_access,
      TashApp.Init'access);
end tash;
```

Figure 10

This design has been faithfully duplicated in the Ada binding by the use of the Ada 95 access to subprogram capability:

```
type AppInitProc_Ptr is access
function (
   Interp : in Interp_Ptr) return
C.Int;
pragma Convention (C,
AppInitProc_Ptr);
```

The Ada subprogram used to initialize the TASH Tcl interpreter for this situation is shown in Figure 11.

```
function Init (Interp : in
Tcl.Interp_Ptr)
   return C.Int is

begin -- Init

   if Tcl.Init(interp) = Tcl.ERROR then
      return Tcl.ERROR;
   end if;

   Tcl.SetVar(interp, "tcl_rcFileName",
      "~/.tashrc", Tcl.GLOBAL_ONLY);
   return Tcl.OK;

end Init;
```

Figure 11

This actual subprogram would be passed to a Tcl function as follows:

```
Tcl.Main (Argc,
   Argv(Argv'first)'unchecked_access,
   Init'access);
```

TWASH: Tcl Windowing Ada SHell

TWASH is a version of the TASH program which adds the Tk extension. Although it does not provide an Ada binding to Tk, it does allow a Tcl/Tk procedure to be written in Ada. Future plans for this Tcl/Tk binding include providing the capability to use Tk directly from Ada code, similar to the way the **`freq`** program of Appendix B uses Tcl.

Colorado Adatcl

The University of Colorado's Arcadia project [3] has implemented Adatcl [4], an Ada binding to Tcl. Adatcl predates TASH, but has had little influence other than to motivate work on a more complete binding. This section describes the differences between Adatcl and TASH.

Adatcl provides a mutex feature which serializes calls to the Tcl C library. TASH does not. It is clear that this will be necessary if Tcl calls will be made from multiple tasks. This might be especially useful in a situation where multiple Tcl interpreters are needed. This capability has been postponed for a future enhancement.

TASH was built from the ground up for Ada 95 and is not based on an earlier Ada 83 version. It should be more portable than Adatcl because it takes better advantage of new features in Ada 95. See the section on Ada 95 features used in this binding for specifics.

TASH is a more complete implementation. It includes all public Tcl C functions declared in **`tcl.h`** and is based on the latest tcl7.5a2 version.

TASH uses generics to implement ClientData so that an Ada programmer need not use the address attribute or unchecked conversion to manipulate data in hash tables et. al. See the Client Data section for more details.

TASH provides both a "thin" C-style binding as well as an Ada programmer-friendly "thick" binding.

Tasking not used

The Colorado Adatcl uses a mutex semaphore to serialize Ada calls to C functions. This prevents potential problems which may occur as a result of interrupting calls to malloc.

The TASH binding does nothing special to protect calls to C functions. This may be a problem when the binding is called from more than one task. A

future release will use protected types to address this issue.

Ada 95 features used

Most obvious among the Ada 95 features used was the Interfaces.C family of packages. These packages provide C-compatible data types for int, char *, and many others.

Pragma Import provides the capability to call a non-Ada subprogram while assuring compatible argument passing conventions and exerting control over the external name and the link name. This was used to interface to all functions in the Tcl library.

Pragma Convention was used to implement function pointer typedefs as subprogram access types for callbacks from C to Ada. It was also used to assure compatible record and array layout.

Access to subprograms was used for implementing function pointer typedefs.

Access to named objects was used for passing arguments to C functions which require pointers to named objects.

Altogether, the availability of these features in Ada 95 promise to make this binding more portable than one done in Ada 83.

No object-oriented features were used since Tcl is not an object-oriented system.

Testing

Comprehensive testing of this binding has not yet been completed.

All examples in [2], chapters 30 through 32.2 have been implemented in package TestApp (included in the TASH software distribution) and successfully tested. This includes *eq*, *concat*, *list*, *sum*, and *expr* commands and an object-oriented counter.

The test scripts in the Tcl distribution were executed successfully except those which require the additional implementation of the commands in tclTest.c.

Future testing plans include implementation of an Ada version of tclTest.c, completion of all examples in [2] as well as testing of all other functions in the binding not already covered.

Sample programs

One goal of TASH is to allow Ada to be used in place of C to implement Tcl commands. Appendix A contains a complete example extracted from the test software included with the TASH distribution. This program implements one Tcl command in Ada: the eq command defined in [2], section 30.2.

The second goal of TASH is to make the Tcl library functions available for use in Ada programs in which use of Tcl as a scripting language is not necessarily a requirement but where such features as string and list handling, regular expression matching, and hash tables is needed. Appendix B contains a complete Ada program which implements the `freq` script of Figure 1 by using Tcl library functions.

How to get and install TASH

Get TASH via anonymous ftp from **`ocsystems.com`**. It is the file, **`tash1.1b1.tar.gz`** in the directory, **`/xada/tash`**.

Uncompress and extract it from the tar archive:

```
gzcat tash1.1b1.tar.gz | tar xvf -
```

Then, follow these steps to build and test it:

1. Modify the file, "**`Makefile.common`**" in the **`tash1.1b1`** directory to reference the correct location of your local Tcl C library, **`libtcl.a`**.
2. Type "**`make`**" in the **`tash1.1b1`** directory. This executes a make in each of three subdirectories to build the Ada/Tcl interface, the Tcl Ada Shell program, **`tash`**, the **`freq`** demo program, and a test program.
3. Test that the system was correctly built by changing to the **`tash1.1b1/test`** directory and executing "**`make test`**."
4. Try the **`freq`** demo by executing "**`make test`**" in the **`tash1.1b1/demos`** directory. To compare the execution time of Tcl versus Ada **`freq`**, execute "**`make time`**."

Future plans

Plans for the immediate future are to:

- finish implementation of Tcl command procedure examples from [2].
- implement Ada version of tclTest.c. This program is included with the Tcl distribution; it contains extra Tcl command procedures for testing Tcl's C interfaces.
- perform comprehensive testing of all subprograms in Ada/Tcl interface package.

Additional plans include preparation of Ada interfaces for the Tk and TclX extensions and usage

of protected types to access multiple interpreters and serialize calls to C code.

References

1. Gart, M., Interfacing Ada to C – Solutions to Four Problems, *TriAda '95 Proceedings*, ACM Press, New York, N.Y., 1995.
2. Ousterhout, J., *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, Mass, 1994.
3. Arcadia Project, University of Colorado, http://www.cs.colorado.edu/homes/arcadia/public_html/.
4. Adatcl7.3, Arcadia Project, University of Colorado, http://www.cs.colorado.edu/~arcadia/Software/adatcl.html.
5. Wall, Larry and Schwartz, Randy L., *Programming Perl*, O'Reilly & Associates, Sebastopol, CA, 1991.

Appendix A

The following code defines a full Tcl application which contains one new Tcl command, “eq.” The package, TestApp, contains the Ada code of the new Tcl command as well as the required application-specific Tcl initialization function, Init.

```
with Interfaces.C;
with Tcl;

package TestApp is

   package C renames Interfaces.C;

   function Init (
         Interp : in Tcl.Interp_Ptr)
      return C.Int;
   pragma Convention (C, Init);

end TestApp;

---------------------------------------
---

with Ada.Strings.Fixed;
with C_Aux;
with Text_IO;
with Unchecked_Deallocation;

package body TestApp is

   function "+" (Left, Right : in C.Int)
      return C.Int   renames C."+";
   function "=" (Left, Right : in C.Int)
      return Boolean renames C."=";
   function "=" (
      Left, Right : in
C.Strings.Chars_Ptr)
      return Boolean renames C_Aux."=";

   package CreateCommands is new
      Tcl.Generic_CreateCommands
(Integer);

   function EqCmd (
      ClientData : in Integer;
      Interp     : in Tcl.Interp_Ptr;
      Argc       : in C.Int;
      Argv       : in
C_Aux.Chars_Ptr_Ptr)
         return C.Int is

   -- Compares two arguments for
equality
   -- using string comparision.
   -- Returns 1 if equal, 0 if not.

      Vector : C_Aux.Arg_Vector
(1..Argc);
   begin -- EqCmd
      if Argc /= 3 then
         Tcl.SetResult (
            Interp, "wrong # args");
         return Tcl.ERROR;
      end if;
      Vector := C_Aux.Argv.Value (
         Argv, C.Ptrdiff_t(Argc));
      if Vector(Vector'first+1) =
            Vector(Vector'first+2)) then
         Tcl.SetResult (Interp, "1");
      else
         Tcl.SetResult (Interp, "0");
      end if;
      return Tcl.OK;
   end EqCmd;
   pragma Convention (C, EqCmd);

   function Init (
      Interp : in Tcl.Interp_Ptr)
      return C.Int is
   begin -- Init
      if Tcl.Init(interp) = Tcl.ERROR
then
         return Tcl.ERROR;
      end if;
      CreateCommands.CreateCommand (
         interp, "eq", EqCmd'access,
         0, NULL);
      Tcl.SetVar(interp,
"tcl_rcFileName",
         "~/.tashrc", Tcl.GLOBAL_ONLY);
      return Tcl.OK;
   end Init;

end TestApp;

---------------------------------------
---

with Ada.Command_Line;
with C_Aux;
with Interfaces.C.Strings;
with Tcl;
with TestApp;

procedure TaShTest is -- Tcl Ada SHell
Test

   package C renames Interfaces.C;

   function "+" (Left, Right : in C.Int)
      return C.Int renames C."+";

   Argc : C.Int := C.Int (
      Ada.Command_Line.Argument_Count) +
1;
   Argv : C_Aux.Arg_Vector(1..Argc);

begin -- TaShTest

   -- Get command-line arguments and put
   -- them into C-style "argv," as
required
   -- by Tcl.Main.
   C_Aux.Get_Argv (Argv);

   -- Start Tcl
   Tcl.Main (Argc,
      Argv(Argv'first)'unchecked_access,
      TestApp.Init'access);

end TaShTest;
```

Appendix B

The following code is a complete Ada program which implements the `freq` script of Figure 1 by using Tcl library functions. In this case, there is no need to create a Tcl interpreter since none of the Tcl library functions used require it. The Ada version is approximately 75 lines of code while the Tcl version is 10. The extra effort of recoding into Ada yields a 10 to 1 improvement in speed using GNAT 2.07 on a Sun Sparc2 with SunOS 4.1.3.

```
with C_Aux;
with Interfaces.C.Strings;
with Tcl;
with Text_IO;

procedure Freq is -- Frequency counter

   package C renames Interfaces.C;

   function "=" (Left, Right : in
      Tcl.Integer_Hash.HashEntry_Ptr)
      return Boolean renames
      Tcl.Integer_Hash."=";
   function "=" (Left, Right : in C.Int)
      return Boolean renames C."=";

   Line         : C.Strings.Chars_Ptr;
   Freq_Count   : Integer;
   Item         : C.Strings.Chars_Ptr;
   Freq_Hash    : aliased
      Tcl.Integer_Hash.HashTable_Rec;
   Entry_Ptr    :
      Tcl.Integer_Hash.HashEntry_Ptr;
   Is_New_Entry : aliased C.Int;
   Search       : aliased
      Tcl.Integer_Hash.HashSearch_Rec;

   procedure Get_Line (
      Line : in out C.Strings.Chars_Ptr)
is
   -- This procedure gets a line from
   -- standard input and converts it to
a
   -- "C" string.
      Input_Line : String (1..1024);
      Length     : Natural;
   begin -- Get_Line
      Text_IO.Get_Line (
         Input_Line, Length);
      C.Strings.Free (Line);
      Line := C.Strings.New_String (
         Input_Line (1..Length));
   end Get_Line;

begin -- Freq

   -- create a hash table for holding
   -- frequency counts
   Tcl.Integer_Hash.InitHashTable (
      Freq_Hash'unchecked_access,
      Tcl.STRING_KEYS);

   -- read lines from standard input
until
   -- end of file encountered
   while not Text_IO.End_of_File loop
      Get_Line (Line);
      -- create (or find, if already
      -- created) an entry for this line
      Entry_Ptr :=

Tcl.Integer_Hash.CreateHashEntry (
            Freq_Hash'unchecked_access,
            Line,

Is_New_Entry'unchecked_access);
      if Is_New_Entry = 1 then
         Freq_Count := 1;
      else
         -- get the frequency count from
         -- the hash
         Freq_Count :=

Tcl.Integer_Hash.GetHashValue (
               Entry_Ptr) + 1;
      end if;
      -- Store the updated frequency
count
      -- in the table.
      -- WARNING: We take advantage of
the
      -- fact that an integer is the
same
      -- size as a C pointer and store
the
      -- count in the table, rather than
a
      -- pointer to it.
      Tcl.Integer_Hash.SetHashValue (
         Entry_Ptr, Freq_Count);
   end loop;

   -- iterate through every item and
print
   -- it and its frequency count
   Entry_Ptr :=
      Tcl.Integer_Hash.FirstHashEntry (
         Freq_Hash'unchecked_access,
         Search'unchecked_access);
   while Entry_Ptr /= Null loop
      Freq_Count :=
         Tcl.Integer_Hash.GetHashValue (
            Entry_Ptr);
      Item :=
Tcl.Integer_Hash.GetHashKey (
         Freq_Hash'unchecked_access,
         Entry_Ptr);
      Text_IO.Put_Line (C_Aux.Value
(Item)
         & Integer'image (Freq_Count));
      Entry_Ptr  :=
         Tcl.Integer_Hash.NextHashEntry
(
            Search'unchecked_access);
   end loop;

   -- delete the frequency counter
   -- hash table
   Tcl.Integer_Hash.DeleteHashTable (
      Freq_Hash'unchecked_access);

end Freq;
```