Tcl/Tk Engineering Manual

John K. Ousterhout

Sun Microsystems, Inc.
john.ousterhout@eng.sun.com

1. Introduction

This is a manual for people who are developing C code for Tcl, Tk, and their extensions and applications. It describes a set of conventions for writing code and the associated test scripts. There are three reasons for the conventions. First, the conventions ensure that certain important things get done; for example, every procedure must have documentation that describes each of its arguments and its result, and there must exist test scripts that exercise every line of code. Second, the conventions guarantee that all of the Tcl and Tk code has a uniform style. This makes it easier for us to use, read, and maintain each other's code. Third, the conventions help to avoid some common mistakes by prohibiting error-prone constructs such as ++ and −− operators in the middle of expressions.

Most of the conventions originated in the Sprite operating system project at U.C. Berkeley. At the beginning of the Sprite project my students and I decided that we wanted a uniform style for our code and documentation, so we held a series of meetings to choose the rules. The result of these meetings was a document called *The Sprite Engineering Manual*. None of us was completely happy with all the rules, but we all managed to live by them during the project and I think everyone was happy with the results. When I started work on Tcl and Tk, I decided to stick with the Sprite conventions. This document is based heavily on *The Sprite Engineering Manual*.

There are few things that I consider non-negotiable for Tcl and Tk, but the contents of this manual are one of them. I don't claim that these conventions are the best possible ones, but the exact conventions don't really make that much difference. The most important thing is that we all do things the same way. Given that the core Tcl and Tk code follows the conventions, changing the rules now would cause more harm than good.

Please write your code so that it conforms to the conventions from the very start. For example, don't write comment-free code on the assumption that you'll go back and put the comments in later once the code is working. This simply won't happen. Regardless of how good your intentions are, when it comes time to go back and put in the comments you'll find that you have a dozen more important things to do; as the body of uncommented code builds up, it will be harder and harder to work up the energy to go back and fix it all. One of the fundamental rules of software is that its structure only gets worse over time; if you don't build it right to begin with, it will never get that way later. When I write code I typically write the procedure headers for a whole file before I fill in any of the bodies.

The rest of this document consists of 8 major parts. Section 2 discusses the overall structure of a package and how to organize header files. Section 3 describes the structure of a C code file and how to write procedure headers. Section 4 desribes the Tcl/Tk naming conventions. Section 5 presents low-level coding conventions, such as how to indent and where to put curly braces. Section 6 contains a collection of rules and suggestions for writing comments.

Section 7 describes how to write and maintain test suites. Section 8 describes how to make code portable without making it unreadable too. Section 9 contains a few miscellaneous topics, such as keeping a change log.

2. Packages and header files

Tcl applications consist of collections of *packages*. Each package provides code to implement a related set of features. For example, Tcl itself is a package, as is Tk; various extensions such as Tcl-DP, TclX, Expect, and BLT are also packages. Packages are the units in which code is developed and distributed: a single package is typically developed by a single person or group and distributed as a unit. One of the best things about Tcl is that it is possible to combine many independently-developed packages into a single application; packages should be designed with this in mind. This section describes the file structure of packages with an emphasis on header files; later sections discuss conventions for code files. You may also wish to review Chapter 31 of my book *Tcl and the Tk Toolkit* for additional information on packages, such as how to interface them to the rest of an application.

2.1 Package prefixes

Each package has a unique short *prefix*. The prefix is used in file names, procedure names, and variable names in order to prevent name conflicts with other packages. For example, the prefix for Tcl is `tcl`; Tcl's exported header file is called `tcl.h` and exported procedures and variables have names like `Tcl_Eval`.

2.2 Version numbers

Each package has a two-part version number such as 7.4. The first number (7) is called the major version number and the second (4) is called the minor version number. The version number changes with each public release of the package. If a new release contains only bug fixes, new features, and other upwardly compatible changes, so that code and scripts that worked with the old version will also work with the new version, then the minor version number increments and the major version number stays the same (e.g., from 7.4 to 7.5). If the new release contains substantial incompatibilities, so that existing code and scripts will have to be modified to run with the new version, then the major version number increments and the minor version number resets to zero (e.g., from 7.4 to 8.0).

2.3 Overall structure

A package typically consists of several code files, plus at least two header files, plus additional files for building and configuring the package, such as a `Makefile.in` and a `configure.in` file for the `autoconf` program. The header files for a package generally fall into the following categories:

- A *package header file*, which is named after the package, such as `tcl.h` or `tk.h`. This header file describes all of the externally-visible features of the package, such as procedures, global variables, and structure declarations. The package header file is eventually installed in a system directory such as `/usr/local/include`; it is what clients of the package `#include` in their C code. As a general rule of thumb, the package header file should define as few things as possible: it's very hard to change an exported feature since it breaks client code that uses the package, so the less you export, the easier it will be to make changes to the package. Thus, for example, try not to make the internal fields of structures visible in package header files.
- An *internal header file*, which is typically `#included` by all of the C files in the package. The internal header file has a name like `tclInt.h` or `tkInt.h`, consisting of the the package prefix followed by `Int.h`. The internal header file describes features that are used

in multiple files within the package but aren't exported out of the package. For example, key package structures and internal utility procedures are defined in the internal header file. The internal header file should also contain `#includes` for other headers that are used widely within the package, so they don't have to be included over and over in each code file. As with the package header, the internal header file should be as small as possible: structures and procedures that are only used in a single C file in the package should not appear in it.

- A *porting header file*, which contains definitions that hide the differences between the systems on which the package can be used.The name of the porting header should consist of the package prefix follwed by `Port.h`, such as `tclPort.h`.
- Other internal header files for various subpackages within the package. For example, there is a file `tkText.h` in Tk that is shared among all the files that implement text widgets and another file `tkCanvas.h` that is shared among all the files implementing canvases.

I recommend having as few header files as possible in each package. In almost all cases a package header file, a single internal header file, and a porting header file will be sufficient, and in many cases the porting header file may not be necessary. The internal header file should automatically `#include` the package header file and perhaps even the porting header file, so each C file in the package only needs to `#include` one or at most two header files. I recommend keeping the porting header separate from the internal header file in order to maintain a clean separation between porting code and the rest of the module. Other internal headers should only be necessary in unusual cases, such as the Tk text and canvas widgets (each of `tkText.h` and `tkCanvas.h` is many hundred lines long, due to the complexity of the widgets, and they are needed only in the source files that implement the particular widgets, so I thought it would be easier to manage these headers separately from `tkInt.h`). If you have lots of internal header files, such as one for each source file, then you will end up with lots of `#include` statements in each C file and you'll find that either (a) you `#include` every header in every C file (in which case there's not much advantage to having the separate `.h` files) or (b) you are constantly adding and deleting `#include` statements as you modify source files.

2.4 Header file structure

Figure 1 illustrates the format of a header file. Your header files should follow this structure exactly: same indentation, same order of information, and so on. To make this as easy as possible, the directory `engManual` in the Tcl source tree contains templates for various pieces of source files. For example, the file `proto.h` contains a template for a header file; there are also templates for code files and procedure headers. You should be able to set up your editor to incorporate the templates when needed, then you can modify them for the particular situation in which they are used. This should make it easy for you to conform to the conventions without a lot of typing overhead.

Each header file contains the following parts, which are labelled in Figure 1:

Abstract: the first few lines give the name of the file plus a short description of its overall purpose.

Copyright notice: this protects the ownership of the file and controls distribution; different notices may be used on different files, depending on whether the file is to be released freely or restricted. The wording in copyright notices is sensitive (e.g. the use of upper case is important) so don't make changes in notices without checking with a legal authority.

Revision string: the contents of this string are managed automatically by the source code control system for the file, such as RCS or SCCS (SCCS is used in the example in the figure). It identifies the file's current revision, date of last modification, and so on.

Multiple include #ifdef: when a large application is developed with many related packages, it is hard to arrange the `#include` statements so that each include file is included exactly once For example, files `a.h` and `b.h` might both include `c.h`, and a particular code file might include both `a.h` and `b.h`. This will cause `c.h` to be processed twice, and could potentially result in compiler errors such as multiply-defined symbols. With the multiple-

Abstract

```
/*
 * tcl.h --
 *
 *      This header file describes the externally-visible facilities
 *      of the Tcl interpreter.
```

Copyright

```
 *
 * Copyright (c) 1987-1994 The Regents of the University of California.
 * Copyright (c) 1994-1996 Sun Microsystems, Inc.
 *
 * See the file "license.terms" for information on usage and redistribution
 * of this file, and for a DISCLAIMER OF ALL WARRANTIES.
 *
```

Revision String

```
 * SCCS: %Z% %M% %I% %E% %U%
 */
```

Multiple-Include #ifdef

```
#ifndef _TCL
#define _TCL
```

Version Defines

```
#define TCL_VERSION "7.4"
#define TCL_MAJOR_VERSION 7
#define TCL_MINOR_VERSION 4

...
```

Structure Declaration

```
/*
 * The structure returned by Tcl_GetCmdInfo and passed into
 * Tcl_SetCmdInfo:
 */

typedef struct Tcl_CmdInfo {
    Tcl_CmdProc *proc;                /* Procedure that implements command. */
    ClientData clientData;               /* ClientData passed to proc. */
    Tcl_CmdDeleteProc *deleteProc;       /* Procedure to call when command
                                          * is deleted. */
    ClientData deleteData;              /* Value to pass to deleteProc (usually
                                          * the same as clientData). */
} Tcl_CmdInfo;

...
```

Variable Declaration

```
EXTERN int              tcl_AsyncReady;

...
```

Procedure Prototype

```
EXTERN int             Tcl_Eval _ANSI_ARGS_((Tcl_Interp *interp, char *cmd));

...
```

Multiple-Include #endif

```
#endif /* _TCL */
```

Figure 1. An example of a header file. The file `engManual/proto.h` contains a template for header files.

include `#ifdef`, plus the matching `#endif` at the end of the file, the header file can be `#include`d multiple times without problems. The symbol `_TCL` is defined the first time the header file is included; if the header is included again the presence of the symbol causes the body of the header file to be skipped. The symbol used in any given header file should be the same as the name of the header file except with the `.h` stripped off, a _ prepended, and everything else capitalized.

Version defines: for each package, three symbols related to the current version number should be defined. The first gives the full version number as a string, and the second and third give the major and minor numbers separately as integers. The names for these symbols should be derived from the package prefix as in Figure 1.

Declarations: the rest of the header file consists of declarations for the things that are exported from the package to its clients. Most of the conventions for coding these declarations will be discussed later. When declaring variables and procedures, use `EXTERN` instead of extern to declare them external. The symbol EXTERN can then be #defined to either extern or extern "C" to allow the header file to be used in both C and C++ programs. The EXTERN symbol also gets defined differently when compiling on PCs to satisfy the needs of that platform. The header file tcl.h contains code to #define the EXTERN symbol; if your header file doesn't #include tcl.h, you can copy the code from tcl.h to your header file.

2.5 _ANSI_ARGS_ prototypes

Procedure prototypes should use the _ANSI_ARGS_ macro as shown in Figure 1. _ANSI_ARGS_ makes it possible to write full procedure prototypes for the normal case where an ANSI C compiler will be used, yet it also allows the file to be used with older non-ANSI compilers. To use _ANSI_ARGS_, specify the entire argument list, including parentheses, as an argument to the _ANSI_ARGS_ macro; _ANSI_ARGS_ will evaluate to either this argument list or (), depending on whether or not an ANSI C compiler is being used. The _ANSI_ARGS_ macro is defined in tcl.h.

In the argument lists in procedure prototypes, be sure to specify names for the arguments as well as their types. The names aren't required for compilation (for example, the declaration for Tcl_Eval could have been written as

```
EXTERN int          Tcl_Eval _ANSI_ARGS_((Tcl_Interp *, char *));
```

in Figure 1) but the names provide additional information about the arguments.

3. How to organize a code file

Each source code file should contain a related set of procedures, such as the implementation of a widget or canvas item type, or a set of procedures to implement hash tables. Before writing any code you should think carefully about what functions are to be provided and divide them up into files in a logical way. The most manageable size for files is usually in the range of 500-2000 lines. If a file gets much larger than this, it will be hard to remember everything that the file does. If a file is much shorter than this, then you may end up with too many files in a directory, which is also hard to manage.

Code files are divided into pages separated by formfeed (control-L) characters. The first page of the file is a header page containing information that is used throughout the file. Each additional page of the file contains one procedure. This approach has two advantages. First, when you print a code file each procedure header will start at the top of the page, which makes for easier reading. Second, you can browse through all of the procedures in a file by searching for the formfeed characters.

3.1 The file header page

The first page of a code file is a *header page*. It contains overall information that is relevant throughout the file, which consists of everything but the definitions of the file's procedures. The header page typically has six parts, as shown in Figure 2:

Abstract: the first few lines give the name of the file and a brief description of the overall functions provided by the file, just as in header files.

Copyright notice: protects ownership of the file, just as in header files.

Revision string: similar to the revision strings in header files.

Include statements: all of the #include statements for the file should appear on the header file just after the version string. In general there should be very few #include statements in a given code file, typically just for the package's internal header file and porting header file. If additional #includes are needed they should appear in the package's internal header file or porting header file.

Abstract

```
/*
 * tclLink.c --
 *
 *      This file implements linked variables (a C variable that is
 *      tied to a Tcl variable).  The idea of linked variables was
 *      first suggested by Andreas Stolcke and this implementation is
 *      based heavily on a prototype implementation provided by
 *      him.
 *
```

Copyright

```
 * Copyright (c) 1993 The Regents of the University of California.
 * Copyright (c) 1994-1996 Sun Microsystems, Inc.
 *
 * See the file "license.terms" for information on usage and redistribution
 * of this file, and for a DISCLAIMER OF ALL WARRANTIES.
 *
```

Revision String

```
 * SCCS: %Z% %M% %I% %E% %U%
 */
```

Includes

```
#include "tclInt.h"
```

Declarations

```
/*
 * For each linked variable there is a data structure of the following
 * type, which describes the link and is the clientData for the trace
 * set on the Tcl variable.
 */

typedef struct Link {
    Tcl_Interp *interp;         /* Interpreter containing Tcl variable. */
    char *addr;                 /* Location of C variable. */
    int type;                   /* Type of link (TCL_LINK_INT, etc.). */
    int writable;               /* Zero means Tcl variable is read-only. */
    union {
        int i;
        double d;
    } lastValue;                /* Last known value of C variable;  used to
                                 * avoid string conversions. */
} Link;
```

Prototypes

```
/*
 * Prototypes for procedures referenced only in this file:
 */

static char *           LinkTraceProc _ANSI_ARGS_((ClientData clientData,
                            Tcl_Interp *interp, char *name1, char *name2,
                            int flags));
static char *           StringValue _ANSI_ARGS_((Link *linkPtr,
                            char *buffer));
```

Figure 2. An example of a header page. The file `engManual/proto.c` contains a template for a header page.

Declarations: any structures used only in this file should be declared on the header page. In addition, if the file defines any static or global variables then they should be declared on the header page. This makes it easy to tell whether or not a file has static variables, which is important if the file is ever used in a multi-threaded environment. Static variables are generally undesirable; avoid them as much as possible.

Prototypes: procedure prototypes for procedures referenced only in this file should appear at the very end of the header page. Use the _ANSI_ARGS_ macro described in Section 2.5 and be sure to declare them `static`.

Please structure your header pages in exactly the order given above and follow the syntax of Figure 2 as closely as possible. The file `engManual/proto.c` provides a template for a header page.

Source files should never contain `extern` statements. Instead, create header files to hold the `extern` statements and `#include` the header files. This makes code files easier to read and makes it easier to manage the `extern` statements, since they're centralized in `.h` files

```
/*
 *----------------------------------------------------------------------
 *
 * Tcl_GetInt --
 *
 *      Given a string, produce the corresponding integer value.
 *
 * Results:
 *      The return value is normally TCL_OK;  in this case *intPtr
 *      will be set to the integer value equivalent to string.  If
 *      string is improperly formed then TCL_ERROR is returned and
 *      an error message will be left in interp->result.
 *
 * Side effects:
 *      None.
 *
 *----------------------------------------------------------------------
 */

int
Tcl_GetInt(interp, string, intPtr)
    Tcl_Interp *interp;         /* Interpreter to use for error reporting. */
    char *string;               /* String containing a (possibly signed)
                                 * integer in a form acceptable to strtol. */
    int *intPtr;                /* Place to store converted result. */
{
    ...
}
```

Figure 3. The header comments and declaration for a procedure. The file `engManual/prochead` contains a template for this information.

instead of spread around dozens of code files. For example, the internal header file for a package has `extern` statements for all of the procedures that are used by multiple files within the package but aren't exported outside it.

3.2 Procedure headers

Each page after the first one in a file should contain exactly one procedure. The page should begin with a *procedure header* that gives overall documentation for the procedure, followed by the declaration and body for the procedure. See Figures 3 and 4 for examples. The header should contain everything that a caller of the procedure needs to know in order to use the procedure, and nothing else. It consists of three parts:

Abstract: the first lines in the header give the procedure's name, followed by a brief description of what the procedure does. This should not be a detailed description of how the procedure is implemented, but rather a high-level summary of its overall function. In some cases, such as callback procedures, I recommend also describing the conditions under which the procedure is invoked and who calls the procedure, as in Figure 4.

Results: this portion of the header describes describes how the procedure affects things that are immediately visible to its caller. This includes the return value of the procedure and any modifications made to the caller's variables via pointer arguments, such as `intPtr` in Figure 3.

Side Effects: the last part of the header describes changes the procedure makes to its internal state, which may not be immediately visible to the caller but will affect later calls to this or other procedures. This section should not describe every internal variable modified by the procedure. It should simply provide the sort of information that users of the procedure need in order to use the procedure correctly. See Figure 4 for an example.

The file `engManual/prochead` contains a template for a procedure header, which you can include from your editor to save typing. Follow the syntax of Figures 3 and 4 exactly (same indentation, double-dash after the procedure name, etc.).

```
/*
 *--------------------------------------------------------------
 *
 * ScaleBitmap --
 *
 *      This procedure is invoked to rescale a bitmap item in a
 *      canvas.  It is one of the standard item procedures for
 *      bitmap items, and is invoked by the generic canvas code,
 *      for example during the "scale" widget command.
 *
 * Results:
 *      None.
 *
 * Side effects:
 *      The item referred to by itemPtr is rescaled so that the
 *      following transformation is applied to all point coordinates:
 *              x' = originX + scaleX*(x-originX)
 *              y' = originY + scaleY*(y-originY)
 *
 *--------------------------------------------------------------
 */

static void
ScaleBitmap(canvasPtr, itemPtr, originX, originY, scaleX, scaleY)
    Tk_Canvas *canvasPtr;               /* Canvas containing rectangle. */
    Tk_Item *itemPtr;                   /* Rectangle to be scaled. */
    double originX, originY;          /* Origin about which to scale rect. */
    double scaleX;                      /* Amount to scale in X direction. */
    double scaleY;                      /* Amount to scale in Y direction. */
{
    ...
}
```

Figure 4. The header for a procedure with side effects.

3.3 Procedure declarations

The procedure declaration should also follow exactly the syntax in Figures 3 and 4. The first line gives the type of the procedure's result. All procedures must be typed: use `void` if the procedure returns no result. The second line gives the procedure's name and its argument list. If there are many arguments, they may spill onto additional lines (see Sections 5.1 and 5.5 for information about indentation). After this come the declarations of argument types, one argument per line, indented, with a comment after each argument giving a brief description of the argument. Every argument must be explicitly declared, and every argument must have a comment.

This form for argument declarations is the old form that predates ANSI C. It's important to use the old form so that your code will compile on older pre-ANSI compilers. Hopefully there aren't too many of these compilers left, and perhaps in a few years we can switch to the ANSI form, but for now it's better to be safe. Every procedure should also have an ANSI-style prototype either on the file's header page or in a header file, so this approach still allows full argument checking.

3.4 Parameter order

Procedure parameters may be divided into three categories. *In* parameters only pass information into the procedure (either directly or by pointing to information that the procedure reads). *Out* parameters point to things in the caller's memory that the procedure modifies. *In-out* parameters do both. Below is a set of rules for deciding on the order of parameters to a procedure:

1. Parameters should normally appear in the order in, in/out, out, except where overridden by the rules below.

2. If there is a group of procedures, all of which operate on structures of a particular type, such as a hash table, the token for the structure should be the first argument to each of the procedures.
3. When two parameters are the address of a callback procedure and a `ClientData` value to pass to that procedure, the procedure address should appear in the argument list immediately before the `ClientData`.
4. If a callback procedure takes a `ClientData` argument (and all callbacks should), the `ClientData` argument should be the first argument to the procedure. Typically the `ClientData` is a pointer to the structure managed by the callback, so this is really the same as rule 2.

3.5 Procedure bodies

The body of a procedure follows the declaration. See Section 5 for the coding conventions that govern procedure bodies. The curly braces enclosing the body should be on separate lines as shown in Figures 3 and 4.

4. Naming conventions

Choosing names is one of the most important aspects of programming. Good names clarify the function of a program and reduce the need for other documentation. Poor names result in ambiguity, confusion, and error. For example, in the Sprite operating system we spent four months tracking down a subtle problem with the file system that caused seemingly random blocks on disk to be overwritten from time to time. It turned out that the same variable name was used in some places to refer to physical blocks on disk, and in other places to logical blocks in a file; unfortunately, in one place the variable was accidentally used for the wrong purpose. The bug probably would not have occurred if different variable names had been used for the two kinds of block identifiers.

This section gives some general principles to follow when choosing names, then lists specific rules for name syntax, such as capitalization, and finally describes how to use package prefixes to clarify the module structure of your code.

4.1 General considerations

The ideal variable name is one that instantly conveys as much information as possible about the purpose of the variable it refers to. When choosing names, play devil's advocate with yourself to see if there are ways that a name might be misinterpreted or confused. Here are some things to consider:

1. Are you consistent? Use the same name to refer to the same thing everywhere. For example, in the Tcl implementation the name `interp` is used consistently for pointers to the user-visible `Tcl_Interp` structure. Within the code for each widget, a standard name is always used for a pointer to the widget record, such as `butPtr` in the button widget code and `menuPtr` in the menu widget code.
2. If someone sees the name out of context, will they realize what it stands for, or could they confuse it with something else? For example, in Sprite the procedure for doing byte-swapping and other format conversion was originally called `Swap_Buffer`. When I first saw that name I assumed it had something to do with I/O buffer management, not reformatting. We subsequently changed the name to `Fmt_Convert`.
3. Could this name be confused with some other name? For example, it's probably a mistake to have two variables `s` and `string` in the same procedure, both referring to strings: it will be hard for anyone to remember which is which. Instead, change the names to reflect their functions. For example, if the strings are used as source and destination for a copy operation, name them `src` and `dst`.

4. Is the name so generic that it doesn't convey any information? The variable `s` from the previous paragraph is an example of this; changing its name to `src` makes the name less generic and hence conveys more information.

4.2 Basic syntax rules

Below are some specific rules governing the syntax of names. Please follow the rules exactly, since they make it possible to determine certain properties of a variable just from its name.

1. Variable names always start with a lower-case letter. Procedure and type names always start with an upper-case letter.

```
int counter;
extern char *FindElement();
typedef int Boolean;
```

2. In multi-word names, the first letter of each trailing word is capitalized. Do not use underscores as separators between the words of a name, except as described in rule 5 below and in Section 4.3.

```
int numWindows;
```

3. Any name that refers to a pointer ends in `Ptr`. If the name refers to a pointer to a pointer, then it ends in `PtrPtr`, and so on. There are two exceptions to this rule. The first is for variables that are opaque handles for structures, such as variables of type `Tk_Window`. These variables are actually pointers, but they are never dereferenced outside Tk (clients can never look at the structure they point to except by invoking Tk macros and procedures). In this case the `Ptr` is omitted in variable names. The second exception to the rule is for strings. We decided in Sprite not to require `Ptr` suffixes for strings, since they are always referenced with pointers. However, if a variable holds a pointer to a string pointer, then it must have the `Ptr` suffix (there's just one less level of `Ptr` for strings than for other structures).

```
TkWindow *winPtr;
char *name;
char **namePtr;
```

4. Variables that hold the addresses of procedures should have names ending in `Proc`. `typedefs` for these variables should also have names ending in `Proc`.

```
typedef void (Tk_ImageDeleteProc)(ClientData clientData);
Tk_ImageDeleteProc *deleteProc;
```

5. `#defined` constants and macros have names that are all capital letters, except for macros that are used as replacements for procedures, in which case you should follow the naming conventions for procedures. If names in all caps contain multiple words, use underscores to separate the words.

```
#define NULL 0
#define BUFFER_SIZE 1024
#define Min(a,b) (((a) < (b)) ? (a) : (b))
```

6. Names of programs (e.g. `wish`), Tcl commands such as `fconfigure`, and keyword arguments to Tcl commands (e.g. Tk configuration options such as `-highlightwidth`) are usually entirely in lower case, in spite of the rules above. The reason for this rule is that these names are likely to typed interactively, and I thought that using all lower case would make it easier to type them. In retrospect I'm not sure this was a good idea; in any case, Tcl procedure and variable names should follow the same rules as C procedures and variables except that the letter after the underscore isn't usually capitalized.

```
set tcl_arrayList {a b c}
proc pkg_mkIndex {...} {
    ...
}
```

4.3 Names reflect package structure

Names that are exported outside a single file must include the package prefix in order to make sure that they don't conflict with global names defined in other packages. The following rules define how to use package prefixes in names:

1. If a variable or procedure or type is exported by its package, the first letters of its name must consist of the package prefix followed by an underscore. Only the first letter of the prefix is ever capitalized, and it is subject to the capitalization rules from Section 4.2. The first letter after the prefix is always capitalized for C procedures and variables but not for Tcl procedures or variables. The first example below shows an exported C variable, and the second shows an exported type and exported procedure.

```
extern int tk_NumMainWindows;
extern Tcl_Interp *Tcl_CreateInterp(void);
```

2. If a module contains several files, and if a name is used in several of those files but isn't used outside the package, then the name must have the package prefix but no underscore. The prefix guarantees that the name won't conflict with a similar name from a different package; the missing underscore indicates that the name is private to the package.

```
extern void TkEventDeadWindow(TkWindow *winPtr);
```

3. If a name is only used within a single procedure or file, then it need not have the module prefix. To avoid conflicts with similar names in other files, variables and procedures declared outside procedures must always be declared `static` if they have no module prefix.

```
static int initialized;
```

4.4 Standard names

The following variable names are used consistently throughout Tcl and Tk. Please use these names for the given purposes in any code you write, and don't use the names for other purposes.

`clientData`	Used for variables of type `ClientData`, which are associated with callback procedures.
`interp`	Used for variables of type `Tcl_Interp`: these are the (mostly) opaque handles for interpreters that are given to Tcl clients. These variables should really have a `Ptr` extension, but the name was chosen at a time when interpreters were totally opaque to clients.
`iPtr`	Used for variables of type `Interp *`, which are pointers to Tcl's internal structures for interpreters. Tcl procedures often have an argument named `interp`, which is copied into a local variable named `iPtr` in order to access the contents of the interpreter.
`nextPtr`	A field with this name is used in structures to point to the next structure in a linked list. This is usally the last field of the structure.
`tkwin`	Used for variables of type `Tk_Window`, which are opaque handles for the window structures managed by Tk.
`winPtr`	Used for variables of type `TkWindow *`, which are pointers to Tk's internal structures for windows. Tk procedures often take an argument named `tkwin` and immediately copy the argument into a local variable named `winPtr` in order to access the contents of the window structure.

```
if (searchPtr->linesLeft <= 0) {
    goto searchOver;
}

/*
 * The outermost loop iterates over lines that may potentially contain
 * a relevant tag transition, starting from the current segment in
 * the current line.
 */

segPtr = searchPtr->nextPtr;
while (1) {
    /*
     * Check for more tags on the current line.
     */

    for ( ; segPtr != NULL; segPtr = segPtr->nextPtr) {
        if (segPtr == searchPtr->lastPtr) {
            goto searchOver;
        }
        ...
    }
}
```

Figure 5. Comments in code have the form shown above, using full lines, with lined-up stars, the `/*` and `*/` symbols on separate lines, and blank separator lines around each comment (except that the leading blank line can be omitted if the comment is at the beginning of a code block).

5. Low-level coding conventions

This section describes several low-level syntactic rules for writing C code. These rules help to ensure that all of the Tcl code looks the same, and they prohibit a few confusing coding constructs.

5.1 Indents are 4 spaces

Each level of indentation should be four spaces. There are ways to set 4-space indents in all editors that I know of. Be sure that your editor really uses four spaces for the indent, rather than just displaying tabs as four spaces wide; if you use the latter approach then the indents will appear eight spaces wide in other editors.

5.2 Code comments occupy full lines

Comments that document code (as opposed to declarations) should occupy full lines, rather than being tacked onto the ends of lines containing code. The reason for this is that side-by-side comments are hard to see, particularly if neighboring statements are long enough to overlap the side-by-side comments. Comments must have exactly the structure shown in Figure 5, including a leading `/*` line, a trailing `*/` line, and additional blank lines above and below. The leading blank line can be omitted if the comment is at the beginning of a block, as is the case in the second comment in Figure 5. Each comment should be indented to the same level as the surrounding code. Use proper English in comments: write complete sentences, capitalize the first word of each sentence, and so on.

5.3 Declaration comments are side-by-side

When documenting the arguments for procedures and the members of structures, place the comments on the same lines as the declarations. Figures 3 and 4 show comments for procedure arguments and Figure 6 shows a simple structure declaration. The format for comments is the same in both cases. Place the comments to the right of the declarations, with all the left edges of all the comments lined up. When a comment requires more than one line, indent the addi-

```
/*
 * The following structure defines a variable trace, which is used to
 * invoke a specific C procedure whenever certain operations are performed
 * on a variable.
 */

typedef struct VarTrace {
    Tcl_VarTraceProc *traceProc;/* Procedure to call when operations given
                                 * by flags are performed on variable. */
    ClientData clientData;      /* Argument to pass to proc. */
    int flags;                  /* What events the trace procedure is
                                 * interested in:  OR-ed combination of
                                 * TCL_TRACE_READS, TCL_TRACE_WRITES, and
                                 * TCL_TRACE_UNSETS. */
    struct VarTrace *nextPtr;   /* Next in list of traces associated with
                                 * a particular variable. */
} VarTrace;
```

Figure 6. Use side-by-side comments when declaring structure members and procedure arguments.

tional lines to the same level as the first line, with the closing `*/` on the same line as the end of the text. For structure declarations it is usually useful to have a block of comments preceding the declaration, as in Figure 6. This comment before the declaration uses the format given in Section 5.2.

5.4 Curly braces: { goes at the end of a line

Open curly braces should not appear on lines by themselves. Instead, they should be placed at the end of the preceding line. Close curly braces always appear as the first non-blank character on a line. Figure 5 shows how to use curly braces in statements such as `if` and `while`, and Figure 6 shows how curly braces should be used in structure declarations. If an `if` statement has an `else` clause then `else` appears on the same line as the preceding `}` and the following `{`. Close curly braces are indented to the same level as the outer code, i.e., four spaces less than the statements they enclose.

The only case where a { appears on a line by itself is the initial { for the body of a procedure (see Figures 3 and 4).

Always use curly braces around compound statements, even if there is only one statement in the block. Thus you shouldn't write code like

```
if (filePtr->numLines == 0) return -1;
```

but rather

```
if (filePtr->numLines == 0) {
    return -1;
}
```

This approach makes code less dense, but it avoids potential mistakes when adding additional lines to an existing single-statement block. It also makes it easier to set breakpoints in a debugger, since it guarantees that each statement on is on a separate line and can be named individually.

There is one exception to the rule about enclosing blocks in `{}`. For `if` statements with cascaded `else if` clauses, you may use a form like the following:

```
if (strcmp(argv[1], "delete") == 0) {
    ...
} else if (strcmp(argv[1], "get") == 0) {
    ...
} else if (strcmp(argv[1], "set") == 0) {
    ...
} else {
    ...
}
```

```
if ((linePtr->position.lineIndex > position.lineIndex)
        || ((linePtr->position.lineIndex == position.lineIndex)
        && ((linePtr->position.charIndex + linePtr->length)
        > position.charIndex))) {
    return;
}
line = Mx_GetLine(newPtr->fileInfoPtr->file,
        linePtr->position.lineIndex, (int *) NULL);
XDrawImageString(mxwPtr->display, mxwPtr->fileWindow,
        mxwPtr->textGc, x, y + mxwPtr->fontPtr->ascent,
        control, 2);
```

Figure 7. Continuation lines are indented 8 spaces.

5.5 Continuation lines are indented 8 spaces

You should use continuation lines to make sure that no single line exceeds 80 characters in length. Continuation lines should be indented 8 spaces so that they won't be confused with an immediately-following nested block (see Figure 7). Pick clean places to break your lines for continuation, so that the continuation doesn't obscure the structure of the statement. For example, if a procedure call requires continuation lines, try to avoid situations where a single argument spans multiple lines. If the test for an `if` or `while` command spans lines, try to make each line have the same nesting level of parentheses if possible. I try to start each continuation line with an operator such as `*`, `&&`, or `||`; this makes it clear that the line is a continuation, since a new statement would never start with such an operator.

5.6 Label indentation matches the following line of code

A label should have the same indentation as the code that immediately follows it, as in the following example:

```
error:
interp->result = "syntax error in expression";
return TCL_ERROR;
```

5.7 Switch statements have two levels of indentation

In a `switch` statement the `case` keywords should be indented one level (4 spaces) from the `switch` statement, and the bodies of the cases should be indented another level from the `case` keywords:

```
switch (src) {
    case 'n':
        dst = '\n';
        break;
    case 't':
        dst = '\t';
        break;
    ...
}
```

5.8 Avoid macros except for simple things

`#define` statements provide a fine mechanism for specifying constants symbolically, and you should always use them instead of embedding specific numbers in your code. However, it is generally a bad idea to use macros for complex operations; procedures are almost always better (for example, you can set breakpoints inside procedures but not in the middle of macros). The only time that it is OK to use `#define`'s for complex operations is if the operations are critical to performance and there is no other way to get the performance (have you measured the performance before and after to be sure it matters?).

When defining macros, remember always to enclose the arguments in parentheses:

```
#define Min(a,b) (((a) < (b)) ? (a) : (b))
```

Otherwise, if the macro is invoked with a complex argument such as `a*b` or `small||red` it may result in a parse error or, even worse, an unintended result that is difficult to debug.

5.9 Parenthesize expressions

Use parentheses around each subexpression in an expression to make it absolutely clear what is the evaluation order of the expression (a reader of your code should not need to remember C's precedence rules). For example, don't type

```
if (x > 22 && y <= 47) ...
```

Instead, type this:

```
if ((x > 22) && (y <= 47)) ...
```

5.10 No assignment operators

Do not use operators with side effects, like =, ++, and --, inside expressions. It is easy to miss these operators when reading code, and this can lead to subtle errors. Thus the following statements are illegal:

```
while ((c = getchar()) != EOF) {
    ...
}
*dst++ = *src++ + 2;
```

Instead, split the operations up into multiple statements with at most one assignment per statement. This will take more lines of code, but the code will be more readable. It is alright to use ++ or -- in an assignment statement as long as it is the only assignment operator in the statement. The above examples can be rewritten as follows:

```
while (1) {
    c = getchar();
    if (c == EOF) {
        break;
    }
    ...
}
*dst = *src + 2;
src++;
dst++;
```

6. Documenting code

The purpose of documentation is to save time and reduce errors. Documentation is typically used for two purposes. First, people will read the documentation to find out how to use your code. For example, they will read procedure headers to learn how to call the procedures. Ideally, people should have to learn as little as possible about your code in order to use it correctly. Second, people will read the documentation to find out how your code works internally, so they can fix bugs or add new features; again, good documentation will allow them to make their fixes or enhancements while learning the minimum possible about your code. More documentation isn't necessarily better: wading through pages of documentation may not be any easier than deciphering the code. Try to pick out the most important things that will help people to understand your code and focus on these in your documentation.

6.1 Document things with wide impact

The most important things to document are those that affect many different pieces of a program. Thus it is essential that every procedure interface, every structure declaration, and every global variable be documented clearly. If you haven't documented one of these things it will be necessary to look at all the uses of the thing to figure out how it's supposed to work; this will be time-consuming and error-prone.

On the other hand, things with only local impact may not need much documentation. For example, in short procedures I don't usually have comments explaining the local variables. If the overall function of the procedure has been explained, and if there isn't much code in the procedure, and if the variables have meaningful names, then it will be easy to figure out how they are used. On the other hand, for long procedures with many variables I usually document the key variables. Similarly, when I write short procedures I don't usually have any comments in the procedure's code: the procedure header provides enough information to figure out what is going on. For long procedures I place a comment block before each major piece of the procedure to clarify the overall flow through the procedure.

6.2 Don't just repeat what's in the code

The most common mistake I see in documentation (besides it not being there at all) is that it repeats what is already obvious from the code, such as this trivial (but exasperatingly common) example:

```
/*
 * Increment i.
 */

i += 1;
```

Documentation should provide higher-level information about the overall function of the code, helping readers to understand what a complex collection of statements really means. For example, the comment

```
/*
 * Probe into the hash table to see if the symbol exists.
 */
```

is likely to be much more helpful than

```
/*
 * Mask off all but the lower 8 bits of x, then index into table
 * t, then traverse the list looking for a character string
 * identical to s.
 */
```

Everything in this second comment is probably obvious from the code that follows it.

Another thing to consider in your comments is word choice. Use different words in the comments than the words that appear in variable or procedure names. For example, the comment

```
/*
 * VmMapPage --
 *
 *      Map a page.
 * ...
```

which appears in the header for the Sprite procedure VmMapPage, doesn't provide any new information. Everything in the comment is already obvious from the procedure's name. Here is a much more useful comment:

```
/*
 * VmMapPage --
 *
 *      Make the given physical page addressable in the kernel's
 *      virtual address space.  This procedure is used when the
 *      kernel needs to access a user's page.
 * ...
```

This comment tells *why* you might want to use the procedure, in addition to *what* it does, which makes the comment much more useful.

6.3 Document each thing in exactly one place

Systems evolve over time. If something is documented in several places, it will be hard to keep the documentation up to date as the system changes. Instead, try to document each major design decision in exactly one place, as near as possible to the code that implements the design decision. For example, put the documentation for each structure right next to the declaration for the structure, including the general rules for how the structure is used. You need not explain the fields of the structure again in the code that uses the structure; people can always refer back to the structure declaration for this. The principal documentation for each procedure goes in the procedure header. There's no need to repeat this information again in the body of the procedure (but you might have additional comments in the procedure body to fill in details not described in the procedure header). If a library procedure is documented thoroughly in a manual entry, then I may make the header for the procedure very terse, simply referring to the manual entry. For example, I use this terse form in the headers for all Tcl command procedures, since there is a separate manual entry describing each command.

The other side of this coin is that every major design decision needs to be documented *at least* once. If a design decision is used in many places, it may be hard to pick a central place to document it. Try to find a data structure or key procedure where you can place the main body of comments; then reference this body in the other places where the decision is used. If all else fails, add a block of comments to the header page of one of the files implementing the decision.

6.4 Write clean code

The best way to produce a well-documented system is to write clean and simple code. This way there won't be much to document. If code is clean, it means that there are a few simple ideas that explain its operation; all you have to do is to document those key ideas. When writing code, ask yourself if there is a simple concept behind the code. If not, perhaps you should rethink the code. If it takes a lot of documentation to explain a piece of code, it is a sign that you haven't found a clean solution to the problem.

6.5 Document as you go

It is extremely important to write the documentation as you write the code. It's very tempting to put off the documentation until the end; after all, the code will change, so why waste time writing documentation now when you'll have to change it later? The problem is that the end never comes – there is always more code to write. Also, the more undocumented code that you accumulate, the harder it is to work up the energy to document it. So, you just write more undocumented code. I've seen many people start a project fully intending to go back at the end and write all the documentation, but I've never seen anyone actually do it.

If you do the documentation as you go, it won't add much to your coding time and you won't have to worry about doing it later. Also, the best time to document code is when the key ideas are fresh in your mind, which is when you're first writing the code. When I write new code, I write all of the header comments for a group of procedures before I fill in any of the bodies of the procedures. This way I can think about the overall structure and how the pieces fit together before getting bogged down in the details of individual procedures.

6.6 Document tricky situations

If code is non-obvious, meaning that its structure and correctness depend on information that won't be obvious to someone reading it for the first time, be sure to document the non-obvious information. One good indicator of a tricky situation is a bug. If you discover a subtle property of your program while fixing a bug, be sure to add a comment explaining the problem and its solution. Of course, it's even better if you can fix the bug in a way that eliminates the subtle behavior, but this isn't always possible.

7. Testing

One of the environments where Tcl works best is for testing. If all the functionality of an application is available as Tcl commands, you should be able to write Tcl scripts that exercise the application and verify that it behaves correctly. For example, Tcl contains a large suite of tests that exercise nearly all of the Tcl functionality. Whenever you write new code you should write Tcl test scripts to go with that code and save the tests in files so that they can be re-run later. Writing test scripts isn't as tedious as it may sound. If you're developing your code carefully you're already doing a lot of testing; all you need to do is type your test cases into a script file where they can be reused, rather than typing them interactively where they vanish after they're run.

7.1 Basics

Tests should be organized into script files, where each file contains a collection of related tests. Individual tests should be based on the procedure `test`, just like in the Tcl and Tk test suites. Here are two examples:

```
test expr-3.1 {floating-point operators} {
    expr 2.3*.6
} 1.38
test expr-3.2 {floating-point operators} {
    list [catch {expr 2.3/0} msg] $msg
} {1 {divide by zero}}
```

`test` is a procedure defined in a script file named `defs`, which is `source`d by each test file. `test` takes four arguments: a test identifier, a string describing the test, a test script, and the expected result of the script. `test` evaluates the script and checks to be sure that it produces the expected result. If not, it prints a message like the following:

```
==== expr-3.1 floating-point operators
==== Contents of test case:

    expr 2.3*.6

==== Result was:
1.39
---- Result should have been:
1.38
---- expr-3.1 FAILED
```

To run a set of tests, you start up the application and `source` a test file. If all goes well no messages appear; if errors are detected, a message is printed for each one.

The test identifier, such as `expr-3.1`, is printed when errors occur. It can be used to search a test script to locate the source for a failed test. The first part of the identifier, such as `expr`, should be the same as the name of the test file, except that the test file should have a `.test` extension, such as `expr.test`. The two numbers allow you to divide your tests into groups. The tests in a particular group (e.g., all the `expr-3.`*n* tests) relate to a single sub-feature, such as a single C procedure or a single option of a Tcl command. The tests should appear in the test file in the same order as their numbers.

The test name, such as `floating-point operators`, is printed when errors occur. It provides human-readable information about the general nature of the test.

Before writing tests I suggest that you look over some of the test files for Tcl and Tk to see how they are structured. You may also want to look at the `README` files in the Tcl and Tk test directories to learn about additional features that provide more verbose output or restrict the set of tests that are run.

7.2 Organizing tests

Organize your tests to match the code being tested. The best way to do this is to have one test file for each source code file, with the name of the test file derived from the name of the source file in an obvious way (e.g. `textWind.test` contains tests for the code in `tkTextWind.c`). Within the test file, have one group of tests for each procedure (for example, all the `textWind-2.`*n* tests in `textWind.test` are for the procedure `TkTextWindowCmd`). The order of the tests within a group should be the same as the order of the code within the procedure. This approach makes it easy to find the tests for a particular piece of code and add new tests as the code changes.

The Tcl test suite was written a long time ago and uses a different style where there is one file for each Tcl command or group of related commands, and the tests are grouped within the file by sub-command or features. In this approach the relationship between tests and particular pieces of code is much less obvious, so it is harder to maintain the tests as the code evolves. I don't recommend using this approach for new tests.

7.3 Coverage

When writing tests, you should attempt to exercise every line of source code at least once. There will be occasionally be code that you can't exercise, such as code that exits the application, but situations like this are rare. You may find it hard to exercise some pieces of code because existing Tcl commands don't provide fine enough control to generate all the possible execution paths (for example, at the time I wrote the test suite for Tcl's dynamic string facility there were very few Tcl commands using the facility; some of the procedures were not called at all). In situations like this, write one or more new Tcl commands just for testing purposes. For example, the file `tclTest.c` in the Tcl source directory contains a command `testdstring`, which provides a number of options that allow all of the dynamic string code to be exercised. `tclTest.c` is only included in a special testing version of `tclsh`, so the `testdstring` command isn't present in normal Tcl applications. Use a similar approach in your own code, where you have an extra file with additional commands for testing.

It's not sufficient just to make sure each line of code is executed by your tests. In addition, your tests must discriminate between code that executes correctly and code that isn't correct. For example, write tests to make sure that the `then` and `else` branches of each `if` statement are taken under the correct conditions. For a loop, run different tests to make the loop execute zero times, one time, and two or more times. If a piece of code removes an element from a list, try cases where the element to be removed is the first element, last element, only element, and neither first element nor last. Try to find all the places where different pieces of code interact in unusual ways, and exercise the different possible interactions.

7.4 Memory allocation

Tcl and Tk use a modified memory allocator that checks for several kinds of memory allocation errors, such as freeing a block twice, failing to free a block, or writing past the end of a block. In order to use this allocator, don't call `malloc`, `free`, or `realloc` directly in C code. Use `ckalloc` instead of `malloc`, `ckfree` instead of `free`, and `ckrealloc` instead of `realloc`. These procedures behave identically to `malloc`, `free`, and `realloc` except that they monitor memory usage. `Ckalloc`, `ckfree`, and `ckrealloc` are actually macros that can be configured with a compiler switch: if `TCL_MEM_DEBUG` is defined, they perform the checks but run more slowly and use more memory; if `TCL_MEM_DEBUG` is not defined, then the macros are just `#defined` to `malloc`, `free`, and `realloc` so there is no penalty in efficiency. I always run with `TCL_MEM_DEBUG` in my development environment and you should too. Official releases typically do not have `TCL_MEM_DEBUG` set.

If you set `TCL_MEM_DEBUG` anywhere in your code then you must set it everywhere (including the Tcl and Tk libraries); the memory allocator will get hopelessly confused if a block of memory is allocated with `malloc` and freed with `ckfree`, or allocated with `ckalloc` and freed with `free`.

There is nothing equivalent to `calloc` in the debugging memory allocator. If you need a new block to be zeroed, call `memset` to clear its contents.

If you compile with `TCL_MEM_DEBUG`, then an additional Tcl command named `memory` will appear in your application (assuming that you're using the standard Tcl or Tk main program). The `memory` command has the following options:

`memory active` *`file`*
: Dumps a list of all allocated blocks (and where they were allocated) to *`file`*. Memory leaks can be tracked down by comparing dumps made at different times.

`memory break_on_malloc` *`number`*
: Enter the debugger after *`number`* calls to `ckalloc`.

`memory info`
: Prints a report containing the total allocations and frees since Tcl began, the number of blocks currently allocated, the number of bytes currently allocated, and the maximum number of blocks and bytes allocated at any one time.

`memory init` *`onoff`*
: If *`onoff`* is `on`, new blocks of memory are initialized with a strange value to help locate uninitialized uses of the block. Any other value for *`onoff`* turns initialization off. Initialization is on by default.

`memory trace` *`onoff`*
: If *`onoff`* is `on`, one line will be printed to stderr for each call to `ckalloc`. Any other value for *`onoff`* turns tracing off. Tracing is off by default.

`memory trace_on_at_malloc` *`number`*
: Arranges for tracing to be turned on after *`number`* calls to `ckalloc`.

`memory validate` *`onoff`*
: If *`onoff`* is `on`, guard zones around every allocated block are checked on every call to `ckalloc` or `ckfree` in order to detect memory overruns as soon as possible. If *`onoff`* is anything other than `on`, checks are made only during `ckfree` calls and only for the block being freed. Memory validation has a very large performance impact, so it is off by default.

The debugging memory allocator is inferior in many ways to commercial products like Purify, so it's worth using one of the commercial products if possible. Even so, please use `ckalloc` and `ckfree` everywhere in your code, so that other people without access to the commercial checkers can still use the Tcl debugging allocator.

7.5 Fixing bugs

Whenever you find a bug in your code it means that the test suite wasn't complete. As part of fixing the bug, you should add new tests that detect the presence of the bug. I recommend writing the tests after you've located the bug but *before* you fix it. That way you can verify that the bug happens before you implement the fix and goes away afterwards, so you'll know you've really fixed something. Use bugs to refine your testing approach: think about what you might be able to do differently when you write tests in the future to keep bugs like this one from going undetected.

7.6 Tricky features

I also use tests as a way of illustrating the need for tricky code. If a piece of code has an unusual structure, and particularly if the code is hard to explain, I try to write additional tests that will fail if the code is implemented in the obvious manner instead of using the tricky approach. This way, if someone comes along later, doesn't understand the documentation for the code, decides the complex structure is unnecessary, and changes the code back to the simple (but incorrect) form, the test will fail and the person will be able to use the test to understand why the code needs to be the way it is. Illustrative tests are not a substitute for good documentation, but they provide a useful addition.

7.7 Test independence

Try to make tests independent of each other, so that each test can be understood in isolation. For example, one test shouldn't depend on commands executed in a previous test. This is important because the test suite allows tests to be run selectively: if the tests depend on each other, then false errors will be reported when someone runs a few of the tests without the others.

For convenience, you may execute a few statements in the test file to set up a test configuration and then run several tests based on that configuration. If you do this, put the setup code outside the calls to the `test` procedure so it will always run even if the individual tests aren't run. I suggest keeping a very simple structure consisting of setup followed by a group of tests. Don't perform some setup, run a few tests, modify the setup slightly, run a few more tests, modify the setup again, and so on. If you do this, it will be hard for people to figure out what the setup is at any given point and when they add tests later they are likely to break the setup.

8. Porting issues

The X Window System, ANSI C, and POSIX provide a standard set of interfaces that make it possible to write highly portable code. However, some additional work will still be needed if code is to port among all of the UNIX platforms. Now that Tcl and Tk are moving from the UNIX world onto PCs and Macintoshes, porting issues have become even more important. This section contains a few tips on how to write code that can run on many different platforms.

8.1 Stick to standards

The easiest way to make your code portable is to use only library interfaces that are available everywhere (or nearly everywhere). For example, the ANSI C library procedures, POSIX system calls, and Xlib windowing calls are available on many platforms; if you code to these standards your packages will be quite portable. Avoid using system-specific library procedures, since they will introduce porting problems.

8.2 Minimize #ifdefs

Although there will be situations where you have to do things differently on different machines, `#ifdefs` are rarely the best way to deal with these problems. If you load up your code with `#ifdef` statements based on various machines and operating systems, the code will turn into spaghetti. `#ifdefs` make code unreadable: it is hard to look at `#ifdef`-ed code and figure out exactly what will happen on any one machine. Furthermore, `#ifdefs` encourage a style where lots of machine dependencies creep all through the code; it is much better to isolate machine dependencies in a few well-defined places.

Thus you should almost never use `#ifdefs`. Instead, think carefully about the ways in which systems differ and define procedural interfaces to the machine-dependent code. Then provide a different implementation of the machine-dependent procedures for each machine. When linking, choose the version appropriate for the current machine. This way all of the machine dependencies for a particular system are located in one or a few files that are totally separate from the machine-dependent code for other systems and from the main body of your code. The only "conditional" code left will be the code that selects which version to link with.

You won't be able to eliminate `#ifdefs` completely, but please avoid them as much as possible. If you end up with code that has a lot of `#ifdefs`, this should be a warning to you that something is wrong. See if you can find a way to re-organize the code (perhaps using the techniques described later in this section) to reduce the number of `#ifdefs`.

8.3 Organize by feature, not by system

Don't think about porting issues in terms of specific systems. Instead, think in terms of specific *features* that are present or absent in the systems. For example, don't divide your code up according to what is needed in HP-UX versus Solaris versus Windows. Instead, consider what features are present in the different systems; for example, some systems have a `waitpid` procedure, while others don't yet provide one, and some systems have ANSI C compilers that support procedure prototypes, while some systems do not.

The feature-based approach has a number of advantages over the system-based approach. First, many systems have features in common, so you can share feature-based porting code among different systems. Second, if you think in terms of features then you can consider each feature separately ("what do I do if there is no `waitpid`?"); this replaces one large problem with several smaller problems that can be dealt with individually. Lastly, the `autoconf` program can be used to check for the presence or absence of particular features and configure your code automatically. Once you've gotten your code running on several different systems, you'll find that many new systems can be handled with no additional work: their features are similar to those in systems you've already considered, so `autoconf` can handle them automatically.

8.4 Use emulation

One of the cleanest ways to handle porting problems is with emulation: assume the existence of certain procedures, such as those in the POSIX standard, and if they don't exist on a given system then write procedures to emulate the desired functionality with the facilities that are present on the system. For example, when Tcl first started being used widely I discovered that many systems did not support the `waitpid` kernel call, even though it was part of the POSIX standard. So, I wrote a `waitpid` procedure myself, which emulated the functionality of `waitpid` using the `wait` and `wait3` kernel calls. The best way to emulate `waitpid` was with `wait3`, but unfortunately `wait3` wasn't available everywhere either, so the emulation worked differently on systems that had `wait3` and those that supported only `wait`. The `autoconf` program checks to see which of the kernel calls are available, includes the emulation for `waitpid` if it isn't available, and sets a compiler flag that indicates to the emulation code whether or not `wait3` is available.

You can also emulate using `#defines` in a header file. For example, not all systems support symbolic links, and those that don't support symbolic links don't support the `lstat` kernel call either. For these systems Tcl uses `stat` to emulate `lstat` with the following statement in `tclUnix.h`:

```
#define lstat stat
```

If a header file is missing on a particular system, write your own version of the header file to supply the definitions needed by your code. Then you can `#include` your version in your code if the system doesn't have a version of its own. For example, here is the code in `tclUnix.h` that handles `unistd.h`, which isn't yet available on all UNIX systems:

```
#ifdef HAVE_UNISTD_H
#include <unistd.h>
#else
#include "compat/unistd.h"
#endif
```

The `configure` script generated by `autoconf` checks for the existence of `unistd.h` in the system include directories and sets `HAVE_UNISTD_H` if it is present. If it isn't present, `tclUnix.h` includes a version from the Tcl source tree.

8.5 Use autoconf

The GNU `autoconf` program provides a powerful way to configure your code for different systems. With `autoconf` you write a script called `configure.in` that describes the porting issues for your software in terms of particular features that are needed and what to do if they aren't present. Before creating a release of your software you run `autoconf`, which processes `configure.in` and generates a shell script called `configure`. You then include `configure` with your distribution.

When it is time to install the distribution on a particular system, the installer runs the `configure` script. `configure` pokes around in the system to find out what features are present, then it modifies the `Makefile` accordingly. The modifications typically consist of compiling additional files to substitute for missing procedures, or setting compiler flags that can be used for conditional compilation in the code.

8.6 Porting header file

In spite of all the above advice, you will still end up needing some conditional compilation, for example to include alternate header files where standard ones are missing or to `#define` symbols that aren't defined on the system. Put all of this code in the porting header file for the package, then `#include` this header file in each of the source files of the package. With this approach you only need to change a single place if you have to modify your approach to portability, and you can see all of the porting issues in one place. You can look at `tclPort.h` and `tkPort.h` for examples of porting header files.

8.7 Do not use bit fields

Bit fields are very hard to use portably (e.g. they may introduce dependencies on bit order within words), so do not use them.

9. Miscellaneous

9.1 Changes files

Each package should contain a file named `changes` that keeps a log of all significant changes made to the package. The `changes` file provides a way for users to find out what's new in each new release, what bugs have been fixed, and what compatibility problems might be introduced by the new release. The `changes` file should be in chronological order. Just add short blurbs to it each time you make a change. Here is a sample from the Tk `changes` file:

```
5/19/94 (bug fix) Canvases didn't generate proper Postscript for
stippled text.

5/20/94 (new feature) Added "bell" command to ring the display's bell.

5/26/94 (feature removed) Removed support for "fill" justify mode
from Tk_GetJustify and from the TK_CONFIG_JUSTIFY configuration
option.  None of the built-in widgets ever supported this mode
anyway.
*** POTENTIAL INCOMPATIBILITY ***
```

The entries in the `changes` file can be relatively terse; once someone finds a change that is relevant, they can always go to the manual entries or code to find out more about it. Be sure to highlight changes that cause compatibility problems, so people can scan the `changes` file quickly to locate the incompatibilities.