a tutorial to

tmk

Hartmut Schirmacher
Max-Planck-Institut für Informatik

tmk – the TCL-based automation software
available from `www.tmk-site.org`

Contents

CHAPTER 1

Introduction

TMK is a flexible and powerful tool for automating all kinds of tasks. It is built on top of TCL, the tool command language created by John Ousterhout [**?**], and combines the syntax and scripting features of this language with the rule-based functionality of MAKE [**?**]. Furthermore, it adds a number of convenience functions, automatic configuration features, and many custom modules for various applications, especially in the context of software development.

Instead of inventing yet another syntax for defining MAKE-like targets and dependencies, TMK is fully embedded in the well-established scripting language TCL. Below is an example defining a rule for compiling C source files into the corresponding object files, in TMK:

```
target {*.o} {$ROOT.c} {
    $C $CFLAGS -o $TARGET $SRC
}
```

For readers experienced with MAKE and/or TCL, this should look somewhat familiar. A classical `Makefile` would not look *so* much different.

Now if you try to imagine a `Makefile` for compiling a complete code directory with multiple source files, automatic dependency update, architecture-dependent output directories, shared libraries from other parts of the project, and external libraries like `QT/QGL` installed somewhere in your system, you would need quote a complicated and large `Makefile`. A typical `TMakefile` for this purpose looks like this:

```
module {cxx qt qgl math pthreads}
set link::PROJLIBS {otherProj/subdir1/subdir2 otherProj/base}
```

As you can see, one strength of TMK is to hide the complicated rule database as well as the usually enourmous mass of site-dependent configuration options, and provide the user/developer with a simple and convenient interface for building files automatically. On the other hand, TMK can easily be customized and adopted for special tasks and needs.

The goal of this tutorial is to teach a novel user most of the capabilities and the basic handling of TCL/TMK. It contains neither a complete TCL language reference [**?**, **?**, **?**], nor a list of all TMK components [**?**].

1.1 Overview

Chapter 2 provides a short introduction to the TCL language and some minor TMK convenience extensions, including strings and basic expressions, lists, variables, procedures, and control flow operations. All readers not familiar with the TCL language are strongly recommended to start with that chapter in order to understand the remaining parts of the tutorial.

Chapter 3 introduces the most important component of the TMK core, which allows defining *targets* and inter-target *dependencies*, similar to the basic functionality of the MAKE tool[**?**]. You need this basically if you want to write your own modules and custom `TMakefile`'s.

Chapter 4 explains how a number of functions can be parameterized and grouped into a so-called *module*. Modules allow to define a number and rules in such way that the actual `TMakefile` contains as few statements as possible.

Chapter 5 shows how to process *directory trees*, group directories in a common *project*, and use project-wide global definitions. This chapter is mandatory for users with more than just a files to be generated by TMK.

Chapter 6 gives insight into one of the major application of TMK, which is handling and compilation of software projects in C/C++ language. The corresponding modules provide a number of functions which handle many everyday sofware development tasks automatically or with minimal cutsomization effort.

1.2 How To Read This Tutorial

This tutorial provides a guided tour through all major topics concerning TMK. In order to understand all further examples, it is strongly recommended to start with Chapter 2, at least with the first three sections. However, readers already familiar with TCL may skip the entire chapter.

- **I Want to Compile A Program**
  If you understand the TCL syntax, you may proceed directly to Chapter 6.

- **I Want To Start A Software Project**
  In addition to Chapter 6, you will also need to know a bit about oranizing a project in a directory tree, which is explained in Chapter 5.

- **I Want To Get Rid Of All My Makefiles**
  If you use MAKE for many different kinds of tasks, you first need to read Chapter 3 and understand how to define targets, rules, and dependencies. Then you should proceed with Chapter 4 in order to learn how to organize classes of rules in common modules.

- **I Want To Learn Some TCL Basics**
  Chapter 2 provides a brief overview of the TCL language, some of its most important predefined functions, and a number of extensions added by TMK. However, if you seriously consider learning TCL, you should consider reading one or more of the many elaborate introductions and tutorials available on the web as well as in every good book store [**?**, **?**, **?**, **?**].

I hope that this tutorial will help you using TCL and TMK for the tasks you need, and that TMK will prove useful and efficient. Enjoy!

Hartmut Schirmacher

CHAPTER 2

TCL/TMK Language Basics

This chapter is supposed to introduce the most relevant components of the TCL language and to some extensions added by TMK. A complete documentation of the TCL language (up to version 7.6) along with a good tutorial can be found in John K. Ousterhout's TCL/TK book [**?**]. There are many more tutorials [**?**, **?**, **?**, **?**]and reference books [**?**, **?**, **?**]available, many of them also including the more recent extensions to the language.

If you want to test some of the examples from this text, you need to install TMK on your system as explained in the TMK reference manual [**?**].

2.1 My First TMakefile

Let's start by doing what many programming tutorials do: write a *hello world* program. TMK is a command line tool which is invoked from the shell and will look for a control file named `TMakefile` in the current working directory. If this file exists, TMK will read it and execute the corresponding commands.

So first let us create a file named `TMakefile` in the current directory, and make it contain the lines

```
# my first tmk script
puts "Hello, World!"
```

TCL programs are strings. If the string starts with a hash character ('#'), everything up to the next newline character is treated as a comment (meaning the string is ignored). If not a comment, the string is parsed *word* by *word*, separated by whitespace characters (space, tab, newline). The first word (e.g. `puts` in our example) is interpreted as the *command name*, and all futher words are treated as command arguments, until either an end-of-line or a semicolon (";") is read.

If properly *quoted*, a word may also contain whitespace or other specially treated characters. Since the `puts` command expects a single argument, the argument `"Hello, World"` is passed as a single quoted word. The `puts` command prints its argument to some file stream defaulting to standard output (represented by the string symbol `stdout`). If you want to print to standard error, the command would read

```
 puts stderr "Hello, World"
```

Now we invoke TMK from within the command shell:

```
tmk
-> tmk: in directory ...
-> Hello, World!
-> tmk: no targets in ...
```

As you can see, TMK does a bit more than just executing the script. Since it usually processes a lot of files and directories, TMK prints some *log messages* (preceeded by the string 'tmk: ') in order to inform the user about where it is and what it does. Since we don't need these log messages right now, we silence TMK using an additional argument:

```
tmk -silent
-> Hello, World!
```

The `-silent` option suppresses log messages, and so TMK just prints what our script produces as output.

If you want TMK to execute a different file instead of `TMakefile`, you can specify the file name using the `-f` option. For example,

```
tmk -f myscript.tmk
```

will read and execute `myscript.tmk` rather than `TMakefile`. If you want to type in your commands interactively rather than reading them from a file, you may use

```
tmk -f -
```

which makes TMK start its *interactive mode* and read commands from the standard input until it encounters an end-of-file (pressing `Ctrl-D` in most command shells) or an `exit` command. In interactive mode, TMK echoes the result of each command line, so that you immediately see the result of such commands as `set`.

If you don't remember the name of a certain command line option, or need to know where on your system you can find the TMK reference manual or tutorial, simply type

```
tmk -help
```

and you will get some hints about the command line options as well as pointers to further documentation.

2.2 Strings, Quoting, Escaping

As we have seen in the previous section, a TCL script is simply a list of whitespace-separated words. TCL interprets the first word as a command name (e.g. `puts`), and then it reads arguments for that command until it encounters either a line break or a semicolon.

```
# multi-line version
puts "A"
puts "B"
puts "C"
# one-line version
puts "A" ; puts "B" ; puts "C"
```

In TCL, everything is represented by a string. Strings do only need to be quoted if they are to represent a single word and contain whitespace characters. If you want to include spaces, tabs, or newlines into a word, you must use either double quotes (”...”) or curly braces ({...}). Note that the word must *start* with the opening brace or quote. Braces and quotes inside of a word are ignored. Braces are parsed *recursively*, so they can be used to define *nested* lists.

```
# two arguments
commandA x y
# one argument, quoted
commandB "x y"
# one argument; quote inside word is ignored
commandB x"y
# one argument, in braces
commandB {x y}
# three arguments, containing nested lists
commandC { {a1 a2 a3} b { c1 c2 } }
```

Since there are several characters which have a special meaning in TCL, you can *escape* their special role by preceeding them with a backslash. If you put a backslash in front of a character without a special meaning, TCL will simply remove the backslash. For example, you can include double quotes or curly braces in a string using the backslash escaping mechanism:

```
# one argument "x y"
commandA {x y}
# two arguments, "{x" and "y}"
commandB \{x y\}
```

Since a line break acts as command separator, you need to escape the newline character if you want to split a command over multiple lines.

```
# split a long command
commandA "first argument" "second argument" \
   "third argument" "fourth argument"
```

If you want to create a string with a number of linebreaks in it (for example, a string representing a TCL/TMK script), the easiest way is to use curly braces, since line breaks are not interpreted as command separators inside of curly braces:

```
 puts {this is a large text which
      expands over several lines!}
-> this is a large text which
->       expands over several lines!
```

2.3 Variable and Command Substitution

You can store an arbitrary string in a *variable* and then use the variable instead of the string. You can set the value of a variable using the `set` command, or append a string using `append`. For expanding the variable's value (called *substitution*), you simply use the variable name, preceeded by a dollar character ('$'):

```
set x "Hello"
-> Hello
append x ", World!"
-> Hello, World!
puts "($x --- $x)"
-> (Hello, World! --- Hello, World!)
```

As you see, variable value expansion also works within double-quoted strings, in contrast to within curly braces. The result of a variable subtitution is always treated as a single word, except that the whole expression is re-evaluated using such commands as `eval` (see TCL documentation).

If you want to place a dollar sign into a string, and *not* cause to expand a variable's value, you can *escape* the dollar sign by preceeding it by a backslash character ("$"):

```
set x "World!"
-> World!
puts "Hello, $x"
-> Hello, World!
puts {Hello, $x}
-> Hello, $x
puts "Hello, \$x"
-> Hello, $x
```

If you need to expand a variable within a string, and if the expanded value if immediately followed by some letter, digit, or underscore, you need to use braces to mark the begin and end of the variable name within the string:

```
set x "Hallo, "
-> Hallo,
puts "$xEcho!"
-> Error: can't read "xEcho": no such variable
puts "${x}Echo!"
-> Hallo, Echo!
```

Similar to expanding variables, you can also substitute the result of a TCL/TMK command into a string. For example the `pwd` command returns a string containing the current working directory. If you want to use this string as an argument to another command, you have to put the `pwd` command (together with its arguments, but this command does not have any) into brackets. As for the variable expansion, command substitution is done in double-quoted strings, but not within a pair of braces.

```
puts pwd
-> pwd
puts [pwd]
-> /home/username/tmk-tutorial
puts "you are in directory [pwd]"
-> you are in directory /home/username/tmk-tutorial
puts {you are in directory [pwd]}
-> you are in directory [pwd]
```

If you want to query whether a certain variable is defined or not, you can simply use the `info exists` command (see example below). Furthermore, you may delete variables explicitly by using `unset`. TMK also defines a command named `set_ifndef` that allows to set a variable only if that variable is not already defined. Furthermore, it also creates the necessary namespace if the variable name contains namespace qualifiers (cf. Sec. 2.11).

```
puts [info exists XYZ]
-> 0
set_ifndef XYZ "Hallo!"
-> Hallo!
puts [info exists XYZ]
-> 1
set_ifndef XYZ "Echo!"
puts $XYZ
-> Hallo!
unset XYZ
puts [info exists XYZ]
-> 0
puts $XYZ
-> can't read "XYZ": no such variable
```

In addition to scalar variables, TCL also supports so-called *associative arrays*. An array is a map containing *key/value* pairs. By specifying the key, you can access the corresponding value. In TCL, both key and value can of course be arbitrary strings. An array is accessed like a scalar variable, except that after the variable name, you have to specify the key. You may not have a scalar variable and an array of the same name.

```
set x("Mona") "Lisa"
set x("Heinz") "Ketchup"
puts $x
-> can't read "x": variable is array
puts $x(Heinz)
-> Ketchup
puts [info exists x(Heinz)]
-> 1
unset x("Ketchup")
puts [array names x]
-> Mona
```

As you can see in the above example, you can do some more interesting things with arrays using the TCL command `array` and its various subcommands. Please refer to the corresponding manual page or the TCL/TK book [**?**] for details.

One interesting and very special array is named `$env`, end reflects the set of active environment variables for the current process and all its future subprocesses. This is discussed later in Sec. 2.8.

2.4 String Operations

TCL provides the `string` command for manipulating strings in different ways. You can find a complete list of all available `string` subcommands in the TCL/TK book [**?**]; we only briefly explain the most commonly used ones here.

The number of characters in a string (including all kind of whitespace etc.) can be queried using the `string length` command. The `range` subcommand subcommand returns the specified character range, with the first character within the string having index 0, and the last character being represented by the symbolic string “end”.

Strings can be converted to upper or lower case using the `toupper` and `tolower` commands, and leading and/or trailing whitespace can be removed using `trim`, `trimleft`, and `trimright`.

```
set x " abc d e f\n "
->  abc d e f
->
puts [string length $x]
-> 11
puts ":[string range $x 2 6]:"
-> :bc d :
puts ":[string range $x 0 end]:"
-> : abc d e f
->  :
puts ":[string trim $x]:"
-> :abc d e f:
```

There are also several tools for finding or matching substrings in strings. The `string first` and `string last` command finds the first or last occurrance of a substring within a string and returns the index of the substring's first character, or `-1` if the substring is not found. `string compare` compares two strings and returns 0 if the strings match exactly, and -1 or 1 if the first string is "smaller" or "greater" than the second in an alphanumerical sense.

```
set x "the quick brown fox"
-> the quick brown fox
set x1 [string first "o" $x]
-> 12
set x2 [string last "o" $x]
-> 17
puts [string range $x $x1 $x2]
-> own fo
puts [string compare "brown" "fox"]
-> -1
```

You can also query whether a string matches a certain *glob pattern*. Glob-style patterns may contain question marks ("?") and stars ("*"), representing arbitrary single characters or a sequence of zero or more characters, respectively. This is similar to glob file matching used in command shells.

```
puts [string match *.h xy.h]
-> 1
puts [string match *.?+? abc.c++]
-> 1
puts [string match *.?? x.c]
-> 0
```

You can formulate even more sophisticated search and replace operations on strings by using *regular expressions*. TCL supports these mainly through two commands, `regexp` and `regsub`, which are beyond the scope of this tutorial.

2.5 List Operations

As we have heard before, a TCL script is simply a list of words. In fact, lists are one of the most prominent features of TCL. A *list* is a possibly empty string containing whitespace-separated *words*. As we have already seen, a word is either a string not containing whitespace, or any other string enclosed in double quotes or curly braces. Here are some examples:

```
# empty lists
set x {} ; set x "" ; set x "  "
# list containing three words
set x {abc "d e f g h" 123}
# list containing a single word
set x {{a b c}}
```

There are several built-in TCL commands for creating, manipulating, and accessing lists. The most often used ones are listed in the command overview 2.5.1.

`list` *`?word0? ?word1? ...`*
: returns a list which contains the specified elements

`lappend` *`list-name ?word0? ?word1? ...`*
: append a number of words to the named variable; changes the variable and returns the retulting value

`llength` *`list`*
: returns number of elements in list

`lindex` *`list index`*
: returns element number *`index`* from *`list`*. The first word in the list has index 0

`lrange` *`list start-index ?end-index?`*
: returns a list containing all elements from *`start-index`* to *`end-index`*. If the end index is omitted or `end`, the last element in the list is assumed

`linsert` *`list index ?word0? ?word1? ...`*
: returns new list, where the elements *`word0`*... are inserted before the element at the specified index; returns new list

`concat` *`listA listB`*
: returns a new list containing the elements of *`listA`*, followed by those of *`listB`*

Commands 2.5.1: basic TCL list operations

Lists can be nested. Each word of a list can in turn be interpreted as a list, according to the quoting rules discussed before. One must pay attention when using whitespace characters and quotes, since TCL tries to preserve each list element exactly as it was.

```
set x "1 2 3"
-> 1 2 3
append x " 4"
-> 1 2 3 4
lappend x " 5"
-> 1 2 3 4 { 5}
lappend x 6 7
-> 1 2 3 4 { 5} 6 7
```

The above example shows the subtle difference between `append` and `lappend`. `append` appends a *string* to a *string*, and does not care about an interpretation of the string's contents. `lappend` treats the first argument as a list, and all further arguments as *words* to be appended to the list. If a word contains whitespace, TCL plays smart and puts braces around the word all by itself, since else the whitespace would get lost when the list is interpreted later.

TCL's list operations allow putting together even complex data structures with only a simple primitive data type. The only difficult thing is to keep in mind the quoting rules, and the fact that TCL only performs a single level of substitutions at a time. Here is another example demonstrating nested lists.

```
puts [llength "1 2"]
-> 2
set x "1 {2a 2b 2c} 3"
-> 1 {2a 2b 2c} 3
puts [llength $x]
-> 3
puts [lindex $x 1]
-> 2a 2b 2c
puts [list [list 1a "1b1 1b2" 1c] 2 3 [list 4a 4b]]
->  {1a {1b1 1b2} 1c} 2 3 {4a 4b}
set x "a b c"
-> a b c
lappend x "d e f"
-> a b c {d e f}
lappend x g h i
-> a b c {d e f} g h i
```

There are even more list operations built into standard TCL, e.g. for searching elements within a list (`lsearch`), or for sorting a list numerically or alphabetically (`lsort`). For more details on these operations, please refer to Chapter 6 of Ousterhout's TCL/TK book [**?**].

TMK defines some additional routines which are especially useful for dealing with lists of filenames or options. These routines deal with *mapping* an expression to each list element (`lmap`), *filtering* only those elements for which some expression becomes true (`lfilter`), and removing certain elements from a list (`lminus` and `lremove`). The syntax of these commands is listed in the command overview 2.5.2.

`lfilter` *`list I-expression`*
: returns new list, containing only those elements for which *`I-expression`* becomes true

`lmap` *`list I-expression`*
: returns new list, where each element is replaced by *`I-expression`* applied to the original element. The *`I-expression`* will be evaluated one extra time, so that the words of the expression will be included as separate list elements, rather than the whole expression as a single element (see example on page 20)

`lmatch` *`list pattern-list`*
: returns new list containing those elements of *`list`* that match any of the glob-style patterns in *`pattern-list`*.

`lminus` *`listA listB`*
: returns new list, containing only those elements of *`listA`* which are not contained in *`listB`*.

`lremove` *`list-name pattern ?nargs?`*
: removes from variable *`list-name`* all those elements which match the specified *`pattern`*. If *`nargs`* is specified, then also remove the *`nargs`* words following each matched element removed.

Commands 2.5.2: TMK's additional list functions

Both `lfilter` and `lmap` use item-dependent *I-expression*. These expressions are evaluated separately for each list item, and may contain special variables which are set to item-dependent values. For example, the variable `$ITEM` will expand to the complete list item, and `$IROOT` will expand to the root name of the list item (which means all characters up to (and not including) the last dot ('.'). Here is an example of what you can do with *I-expressions*:

```
puts [lfilter "100 20 30 44.4" {$ITEM > 30}]
-> 100 44.4
puts [lmap "a b c" {$ITEM $ITEM}]
-> a a b b c c
puts [lmap "a.tif b.tif c.tif" \$IROOT.jpg]
-> a.jpg b.jpg c.jpg
```

Since the special I-expression variables can only be expanded within the command, they have to be escaped when calling the functions, e.g. by putting the I-expression in curly braces, or by escaping each dollar sign with a backslash, as shown in the example above.

Besides the I-expression variables `$ITEM` and `$IROOT`, there are some more which correspond mainly to the different filename operations provided by the TCL `file` command (see [**?**]). Here is a list of the available I-expression variables:

`$ITEM`: complete list item value

`$IBASE`: item's base name starting after the last slash and ending before the last dot; the same as `[file rootname [file tail $ITEM]]`

`$IDIR`: item's directory/path, ending before the last slash

`$IEXT`: item's suffix/extension, starting with the last dot

`$IROOT`: item's root name, ending before the last dot

`$ITAIL`: item's tail, starting after the last slash

The list element removal operations work in a straightforward way. `lminus` literally subtracts the second set of elements from the first list, and returns the remaining elements. `lremove` can be used especially well for removing unwanted flags or files from any list variable. Here are some examples:

```
puts [lminus "a b c a b c d e" "a c e x"]
-> b b d
set x "plane34.gif car1.gif car2.tif car3.bmp bike4.jpg"
-> plane34.gif car1.gif car2.tif car3.bmp bike4.jpg
lremove x car*
-> plane34.gif bike4.jpg
set flags "-woff 13,14 -xy -woff 12 -O2 -DNDEBUG"
lremove flags -woff 1
-> -xy -O2 -DNDEBUG
```

Another very useful pair of commands is `split` and `join`. These two convert between lists and strings in some sense. `split` splits a string using any of the specified characters as separator, and returns a list holding the resulting parts as elements. `join` allows putting together all elements of a list by joining neighbouring elements with a provided string.

```
set x "abc:1,2,3:blabla;xy"
-> abc:1,2,3:blabla;xy
set y [split $x ":;"]
-> abc 1,2,3 blabla xy
puts [split [lindex $y 1] ","]
-> 1 2 3
puts [join $y " - "]
-> abc - 1,2,3 - blabla - xy
```

2.6 Boolean and Numerical Expressions

So far we have treated strings and lists as the primitive data types. For some operations, it is necessary to interpret the value of a certain string as a boolean or numeric value. For this case, TCL defines the `expr` command. `expr` concatenates an arbitrary number arguments to a single list and evaluates this as a numeric expression. An integer value of zero can also be interpreted as a boolean `false`, all other integers are interpreted as `true`.

`expr` returns the resulting numerical value as a string (since this is TCL's only basic data representation). For example, you can use some basic math using integer or floating-point representation:

```
puts "4 + 3*7"
-> 4 + 3*7
puts [expr 4 + 3*7]
-> 25
puts [expr 7.0/3.1]
-> 2.25806451613
```

In order to form boolean expressions, all kinds of boolean operators can be used for comparing numbers or strings, and for building more complex logical expression from simpler ones. The syntax is the same as in the C and C++ language.

```
puts [expr 7<3]
-> 0
puts [expr (7>3) && (4!=5)]
-> 1
puts [expr (4-3==1) && ![string compare xyz xyz]]
-> 1
```

The `string compare` command returns 0 if the two arguments are identical, 1 if the first argument is ”bigger” than the second one in an alphanumerical sense, and -1 else.

2.7 TCL Control Flow

Using the boolean expressions from the previous section, we can have a look at the control flow commands known by TCL. The most important of all these constructs is the `if` command. `if` executes TCL scripts depending on the value of boolean expressions. Here is an example:

```
if {$x == "1"} {
       puts "only a single one."
} elseif {$x == "2"} {
       puts "there are two of them."
} elseif {$x == "3"} {
       puts "if found three!"
} else {
       puts "more than three!!!"
}
```

The `if` expression can contain an arbitrary number of `elseif` branches, and at maximum one `else` branche. Since `if` commands tend to stretch over multiple lines, and since we usually do not want parts of the command to be evaluated before the `if` command is actually executed, it is most convenient to put the expressions and branches in curly braces, as shown in the example. As already mentioned in Section 2.2, line breaks are not interpreted as command separators in curly braces.

Besides the conditional execution of code, TCL also supports all kinds of loops. When working with lists, it is often most convenient to use the `foreach` command, which goes through a list, sets the loop variable to the current list element, and executes a script containing loop variable expressions:

```
foreach x "x y" {
       puts "working on $x"
}
-> working on x
-> working on y
```

Using boolean expression as explained in Section 2.6, one can also use `while` and `for` loops:

```
set x 1
while {$x < 3} {
       puts "this is number $x"
       incr x
}
-> this is number 1
-> this is number 2
for {set x 2} {x > 0} {incr x -1} {
       puts "this is number $x"
}
-> this is number 2
-> this is number 1
```

As you may have guessed from the example, the `incr` command increments the value of a variable by the specified integer amount, with the default argument being 1. When specifying the conditional arguments for `for` and `while`, it is important to quote the boolean expressions so that they will not be immediately replaced by their initial value. The following example would lead to an infinite loop:

```
# bad example - infinite loop
set x 1
while "$x < 2" {
       puts $x
       incr x
}
```

In order to understand why the above example does not work, you must be aware that in the `while` line, the double-quoted expression will first be evaluated to 1, and then the loop iteration will be started with `"1"` being the conditional expression used for testing loop termination.

Inside loop commands, you can use the `break` statement to exit the innermost active loop, and the `continue` statement to continue with the next iteration of the loop.

2.8 Shell Commands and Environment

One of the most important features of a scripting language like TCL is to execute external programs as from the command shell. In TCL, this is realized through the `exec` command:

```
set x [exec echo "Hallo, Echo!"]
puts $x
-> Hallo, Echo!
```

In the above example, the program `echo` is executed and prints a string to the standard output channel. The `exec` command redirects and returns this output, so that the result of the command can be used for further operations. Please note that the first argument to `exec` is interpreted as the command name to be executed, and will be searched in those paths currently contained in the `PATH` environment variable. All further arguments are passed as separate parameters to that command.

Since environment variables often have influence on the behaviour of commands, it is important to have a means for manipulating them from within your TCL/TMK script. This is indeed very easy, since the environment can be accessed through the global array named `$env` (cf. Sec. 2.3). So for example, you can append something to the program search path like this:

```
append env(PATH) ":/home/myself/bin"
```

Environment manipulations have influence on the current process as well as that of any subprocesses, e.g. those created by the `exec` command.

Like in a standard UNIX shell (like `sh`), commands can be combined using so-called *pipes*, and commands can be executed in the *background*, so that TCL will not wait for the command to terminate. Please refer to the TCL/TK book for details.

TMK defines a command called `cmd` which basically does the same as `exec`. In contrast to `exec`, the output of the command is redirected to the real standard output and error channels. Furthermore, unless the `-silent` command line option is used, TMK will echo the issued command to the standard error channel, so that the user sees which shell commands are executed in what order.

```
cmd echo "Hallo, Echo!"
-> echo Hallo, Echo!
-> Hallo, Echo!
```

The arguments are passed to the shell command as they are passed to `exec` or `cmd`. This means that no file pattern matching takes place, e.g. the command `exec echo *.jpg` will simply output the string `*.jpg`. However, the shell-style pattern matching (called *globbing*) can be applied using the `glob` command. `glob` takes a number of patterns and returns a list of all files which match the current pattern. The employable patterns are similar to those of the `string match` command (Section 2.4).

```
cmd ls
-> ls
-> x.tif y.gif a.gif b.jpg c.jpg d.bmp
puts [glob *.jpg *.tif]
-> b.jpg c.jpg x.tif
```

Another important command is `pwd`, which returns the current working directory of the process in which TMK is running.

2.9 File Operations

One of the most important kind of operations are those dealing with files. TCL provides a number of very convenient commands for querying the status of files and directories, or for querying and constructing the different parts of path and file names. Most of these operations are done via the TCL `file` command. The examples below are UNIX-style, but the same kind of operations also work with Windows-style file and path names.

```
puts [file readable /etc/passwd]
-> 1
puts [file isdirectory /etc/passwd]
-> 0
puts [file dirname /etc/passwd]
-> /etc
puts [file tail /etc/passwd]
-> passwd
puts [file extension /etc/init.d]
-> .d
```

Another important means of manipulating paths and filenames are the two commands `join` and `split`, which allow splitting a path so that its components become the elements of a list, and for re-joining the components in order to obtain a path again. Please see Section 2.5, page 21 for details.

TCL defines a lot more file operations, like `size`, `rename`, `copy`, `delete`, or `volume`. There are also commands for constructing paths in a platform-independent way. Please have a look at the TCL/TK book [**?**] or the system manual page for more details on the different options of the `file` command.

TMK defines two additional commands for reading the context of a file and appending in to a string, and for writing back a string into a file. Both commands expect a filename and a variable name.

```
set text "Hello, World!"
write_file /tmp/myfile.txt text
puts [file exists /tmp/myfile.txt]
-> 1
read_file /tmp/myfile.txt new_text
puts $new_text
-> Hello, World!
```

If the specified file does not exist, `read_file` will append an empty string. `write_file` has an additional optional argument. If you specify “append” as third argument (e.g. `write_file xy.txt text append`), the contents of the variable will be appended if the file exists, rather than replacing its previous contents.

2.10 User-defined Functions

Since TCL is a full-fledged scripting language, it also supports user-defined procedures and funtions. The TCL `proc` command takes three arguments: the name of the procedure to be defined, a list of parameter names, and a TCL script. The actual parameters values are assigned to *local variables* that have the specified parameter names. Here is an example:

```
proc OutputTriple { a b c } {
       puts "($a,$b,$c)"
}
OutputTriple 1 2 3
-> (1,2,3)
OutputTriple X abc Y
-> (X,abc,Y)
```

A procedure always returns the return value of its last command. If you want to specify an explicit return value of return in the middle of the procedure body, you can use the `return` command:

```
proc MyFunc {x} {
   if {$x<5} {
      return "smaller than five"
   } else {
      return "greater or equal to five"
   }
}
puts [MyFunc 3]
-> smaller than five
```

As already stated, the actual function call parameters are made available through local variables. All variables that you define within the function body (e.g. using the `set` command) are considered *local*. By default, you do not have access to any variables declared outside the function body.

```
set GLOBAL_VAR "Hallo"
proc MyProc {} {puts $GLOBAL_VAR}
MyProc
-> can't read "GLOBAL_VAR": no such variable
```

If you want to access a *global* variable (e.g. one that is defined outside any function or procedure body), you have to declare a variable name as `global`:

```
proc MyProc {} {
     global GLOBAL_VAR
     puts $GLOBAL_VAR
}
set GLOBAL_VAR "Hallo"
MyProc
-> Hallo
```

Besides using local and global variables, you can also have access to some in-between context level. For example, imagine a procudure `B` called from inside another procedure `A`. If you want to access `A`'s local variables within `B`, you can do this by making those variables known using the `upvar` command. Similarly, you can evaluate expressions in the context of the calling function (or any higher-level context, up to the global context) by using the `uplevel` command. Last, but not least, you can define procedures with a variable number of arguments by using the special argument name `args`, and define default values for some or all arguments. Please refer to chapter 8 of the TCL/TK book [**?**] for details, or to the `proc` manual page.

2.11 Namespaces

So far, we have considered all our global variables and functions to shared the same logical space. However, TCL provides a concept named *namespace* in order to group variables and procedures into different spaces that are independent from each other. Namespaces in TCL are managed dynamically, and you can do a lot of things using the `namespace` TCL command. Here is an example:

```
namespace eval mySpace {
  variable var1 "xyz"
  proc proc1 args {...}
  puts "var1 is $var1"
}
puts "mySpace’s var1 is $mySpace::var1"
```

This code creates a new variable and a procedure, both in a namespace called `myS-pace`. The `variable` works the same way as `set`, but makes sure that the variable is assigned to the current namespace. If we would use `set` instead, and if a global variable of the same name existed, the `set` command would affect the global variable rather than allocating a new one in the current namespace.

As you can see in the example, you can use a namespace variable inside the same namespace as you would use any other variable. This way uses so-called *unqualified names*. Like shown in the last line of the example, you can also address a namespace variable outside the namespace by using *namespace qualifiers*. The double colon `::` is used as the namespace separator, and you can nest namespaces as deeply as you want:

```
namespace eval ns1 {
  namespace eval ns2 {
    namespace eval ns3 {
      variable var1 "xyz"
    }
  }
}
puts "...var1 is $::ns1::ns2::ns3::var1"
```

You may not set a namespace variable before its namespace has been created, but you can create namespaces using `namespace eval` as in the examples above. If you want to make sure that a variable exists before you use it, you can create it using the `variable` statement without specifying a value for the variable:

```
namespace eval xyz {variable x}
lappend xyz::x "hello" "world"
```

An unqualified variable name is first looked up in the currently active namespace, and then in the global namespace (`::`). It is *not* looked up in parent namespaces recursively. Take this example:

```
variable x 10
namespace eval a {
  variable x 20
  puts $x
  namespace eval b {
    puts $x
  }
}
```

So this piece of code produces the results 20 and 10, since in the innermost namespace 'x' refers to the variable in the global namespace.

2.12 Dealing with Runtime Errors

Sometimes, TCL commands may fail, and the TCL shell will terminate with a cryptic error message. If you want to prevent this, you can always "catch" error conditions and prevent TCL from exiting. This is done by the `catch` command, which expects two parameters: the TCL script to be executed, and a variable name for storing the result of the script execution. If `catch` returns `0`, no error has occurred, and you may use the command's result stored in the specified variable. If `catch` returns with a non-null status, you can simply continue with your script, or output the error message stored in the result variable.

```
if [catch {file delete mytext.txt} msg {
  puts stderr "could not delete file ($msg)"
} else {
  puts stdout "file deleted."
}
```

CHAPTER 3

Targets and Dependencies

Targets and dependencies are some of the core features of TMK by which the functionality of MAKE is embedded into the TCL language. They are very popular concepts for doing work only if it is needed, and are commonly used in software development, systems administration, and other software automation areas.

This chapter starts with some simple examples of how to define targts, and then plunges into some advanced features and subtle details, e.g. what exactly happens in the case of many dependencies and multiple rules for the same target, and how exceptions from general rules can be defined.

3.1 Targets and Dependency Chains

A *target* defines how some *target file* can be created from a number of *source files*, or *primary dependencies*. The `target` procedure has three arguments: a list of target patterns, a list of source file expressions, and a TMK script.

```
puts "Hello!"
target hello.y hello.x {
    set txt {}
    read_file "hello.x" txt
    append txt "\nThis is an additional line"
    write_file "hello.y" txt
}
```

The above example states that the target file `hello.y` can be built from the source file `hello.x`. The corresponding *building rule* (the script provided as third argument to the `target` procedure) reads `hello.x` into the variable `$txt`, appends a line of text, and then writes the resulting string into the file `hello.y`.

Now, if you write the above example into a `TMakefile` and invoke `tmk`, you will get the output `Hello!`, but nothing else will happen. This is because you have only *defined* the target, but not told TMK to actually *build* it. This can be done in two different ways, either by specifying the targets to be built on the command line when invoking TMK:

```
tmk hello.y
```

or by declaring it a *default target* using the `build` command in the `TMakefile`:

```
target hello.y [...]
build hello.y
```

Targets specified at the command line override the default targets specified using `build`. The resulting targets that will be built are called *top-level targets*. The actual building process starts *after* the `TMakefile` has been read and processed completely.

For every top-level target to be built, TMK will first perform an *up to date* check. A target is up to date if it exists and if it is newer or of same age as all its *prerequisites*. In our example, this means that `hello.y` must exist and must be newer or of the same age as `hello.x`. If not, the target needs to be updated, and the corresponding rule (the script provided as third argument to the `target` command) will be executed.

In addition to just checking the age of files, each prerequisite is treated as a new *target* recusively. This way you can define arbitrary *dependency chains* that define the order of rule execution.

```
target A {B C} {
  puts "(A <- B C)"
}
target B {D E F G} {
  puts -nonewline "(C <- D E F G) "
}
foreach x {C D E F G} {
  target $x {} "puts -nonewline \"($x)\""
}
build A
```

Running the above example clarifies the order in which dependency chains are processed. TMK is told to build target `A`. For this, it recursively checks `B` and `C`. `B` requires the source files `D,E,F,G`, which are *unconditional* targets, meaning that they can be built without any prerequisites. After doing this, TMK can create `B` from `D E F G`. Then it builds `C`, and then `A` from `B` and `C`. So calling TMK would cause the following output:

```
-> (D) (E) (F) (G) (B <- D E F G) (C) (A <- B C)
```

As you can see, targets are not really required to be real files. You can read more this topic of *pseudo targets* in Section 3.5.

3.2 Target Patterns and T-Expressions

The last example in the previous section showed that if you define a rule for a number of different targets, you must have some means of defining expressions that use the actual target name later when the rule is applied. A better way of doing this than in the past example is to use *T-expressions*. Within a T-expression, you can use a number of special variables that represent different parts of the target name.

Additionally, in order to easily define targets for a whole *class* of files rather than just explicit target names, TMK allows to specify *target patterns*. Here is an example:

```
target *.y {$ROOT.x} {
    set txt {}
    read_file $SRC txt
    append txt "\nThis is an additional line"
    write_file $TARGET txt
}
```

The above example defines a rule for creating arbitrary `.y` files from corresponding `.x` files. The first argument for the `target` command tells that the specified rule can be applied to any target ending in `.y`. Instead of specifying a fix file name, you can use an arbitrary *glob* expression[1], or even a list of glob expressions.

The second argument, which defines the primary dependencies for the target, is a *T-expression*. T-expression are evaluated in the *global* context, implying that all global variables can be accessed without further effort. The expression in the example contains the special target-dependent variable `$ROOT` that expands to the root name of the actual target. T-expressions are very similar to I-expressions discussed in Sec. 2.5, page 20. The variables defined in the context of a T-expression are as follows:

`$TARGET`: complete target name

`$BASE`: base name starting after the last slash, ending before the last dot

`$DIR`: target's directory/path up to the last slash

`$EXT`: target's suffix/extension starting with the last dot

`$ROOT`: root name ending before the last dot

`$TAIL`: target's tail starting after the last slash

The T-expression variables can also be used inside the *rule* that is provided as third argument to the `target` command. Additionally, the special following special variables can be used within the *rule*.

`$SRC`: list of source files (primary dependencies)

`$NEWER_SRC`: list of source files newer than the target **[NOT SUPPORTED YET]**

`$SECONDARY`: list of secondary dependencies **[NOT SUPPORTED YET]**

So you can specify a script for building the target from the source files even though the exact name of these files is not known when specifying the rule. It is important to note that all these target-dependent expressions cannot be evaluated at the time the target is *declared*, since at that point the actual target name is not known in general. So it is imperative to delay their evaluation, e.g. by escaping the T-variables with backslashes, or by putting the expression in curly braces (cf. Sec. 2.2).

An important fact that has not been mentioned yet is that unless specified otherwise, all generated files will be put into an architecture-dependent *target directory*,

[1]Please refer to Sec. 2.4 or the TCL `string` manual page, for details.

beacause the target-dependent variables will automatically be preceeded by a directory name. This special feature of TMK will be ignored here for the sake of clarity, and will be treated later in Sec. 3.7.

In addition to the mentioned T-variables, you can also access variables that contain those parts of the target name that have been matched by one of the *wildcards* ("*" or "?") in the target pattern. The characters matched by the first wildcard is accessible via `$0`, the next via `$1`, and so on. This way it is really simple to define arbitrary rather complicated mappings from target names to source file names:

```
target A*.x? {B$0.y$1 C$0.z[expr $1 - 1]} {
  puts "$TARGET <- $SRC"
}
```

If you run this rather "wild" example for the target `A5.x3`, the rule would print the following message:

```
      A5.x3 <- B5.y3 C5.z2
```

As you can see, you can also include arbitrary function calls like `expr` in the example above, if you need even more complicated target-dependent expressions.

3.3 Multiple Rules and Secondary Dependencies

Since there might be several possibilities of how to generate a certain target, TMK allows to declare multiple rules for the same target. The question now is how TMK determines which rule has to be applied.

For example, you may want to define a rule for compiling source code into an object file. For a C language file, you want to use a compiler called `myC`, and for C++ the compiler is called `myCXX`. C files have the suffix `.c`, and C++ files may end in `.cc`, `.cpp`, `.cxx`, or .C. The name of the output file is specified using the compiler's command line option `-o`.

```
target *.o {$ROOT.c} {
  cmd myC -o $TARGET $SRC
}
foreach suffix {cc cpp cxx C} {
  target *.o \$ROOT.$suffix {
    cmd myCXX -o $TARGET $SRC
  }
}
```

This example[2] defines a number of rules for creating `.o` files. Now, if you want to build some target `x.o`, all these rules have to be taken into account. TMK first finds all rules that match a given target. The order of these rules is *not* defined. For every rule, TMK first tries to build all source files recursively, as explained in Sec. 3.1. If all source files have been brought up to date successfully, TMK executes the current rule for the target and skips all other rules. If at least one of the required source files could not be generated (e.g. does not exist, and there is no rule to build it), TMK skips this rule and tries to apply the next one. If this does not work for any rule, TMK exits with an error.

In addition to the source files or *primary* dependencies, one can also declare any number of *secondary* dependencies for a target. Secondary dependencies are files which are required to make a target, but are not associated with any rule. As a result, any rule will fail if a secondary dependency file is missing or cannot be built.

For example, you can use secondary dependencies control the order in which targets are built, or you can declare files that have to be checked regardless of which rule will be applied. Classically, files included from the source files are declared this way, because they are a sort of "second order" source files for the target.

The TMK command for defining secondary dependencies is called `depend`, and the syntax resembles that of `target`, except that there is no need for a rule. Here is an example `TMakefile` in which the order of execution is controlled through `depend`:

```
target {A1 A2 B} {} {
    puts "This is $TARGET"
}
depend {A*} {B}
build {A1 A2}
```

When running this example, `B` is built before `A1` and `A2`, because the latter two depend on it.

3.4 Manipulating the Up-to-date State

Sometimes it is necessary, or at least convenient, to *pretend* that some file has or has not changed, or to enforce that a certain target be built, even if it would not need to according to the usual rules.

[2]Please note the subtle difference between the source file name components in the second target expression. `$ROOT` is a special T-expression variable that can only be evaluated later when the real target name is known, so it has to be *escaped* using the backslash. `$suffix` is the loop variable and has to be evaluated immediately in order to expand to the desired suffix.

3.4.1 Option `-force`

If you want a certain target to be rebuilt, but TMK thinks that it is up-to-date, you can *force* the building of that target by using the `-force` command line option. When `-force` is active, all top-level targets (default targets, or those specified at the command line)) will be rebuilt, even if they seem up to date. If these targets depend on other targets that have to be rebuilt first, that will be done as well.

3.4.2 Option `-up`

You can also make TMK pretend that a number of targets have just been updated and are thus newer than all other files. The `-up` command line option expects a list of targets which will be marked up-to-date explicitly after reading the `TMakefile`. The list ends with the end of the command line, the next option (starting with '-'), or the explicit end-of-options symbol '`--`'. Here is an example:

```
tmk -up file1 file2 -- file3
```

In this case TMK will build the toplevel target `file3`, and pretend that `file1` and `file2` have just been updated.

3.4.3 Option `-nup`

As the counterpart to the `-up` option, `-nup` pretends that the specified targets have not been touched by marking them internally with the oldest possible time stamp. This works the same way as `-up` , and it also works for a non-existing target: TMK will pretend it is there and has not been updated for years. As a consequence of this, you can even apply `-nup` to pseudo targets such as `ALWAYS_BUILD` (see Sec. 3.5). In that case the targets that exist, but are usually always built, will not be rebuilt.

Of course you can combine all the different target-manipulating command line options like in the following example:

```
tmk -up file1 file2 -nup ALWAYS_BUILD -- myTarget
```

3.4.4 Option `-mfdepend`

In many cases, the contents of the generated files strongly depend on the contents of the `TMakefile`, because certain flags and options that are set in the `TMakefile` will change the behaviour of the programs that generate the files. So a conservative dependency policy would depend each target on the `TMakefile` by which it is built. However, since not every change in the `TMakefile` really changes all the generated files, this is not TMK's default behaviour. You can achieve that additional condition by using the command line option `-mfdepend` , which will add the specified control file (either `TMakefile` or the file specified using the `-f` option) as an additional dependency to every target.

3.4.5 Up-to-date Manipulation Functions

Sometimes it is convenient to control the pretended state of a target from within a `TMakefile` or module. To this end, TMK provides some basic functions for setting and querying a target's state in TMK's internal cache. Those functions are listed in command overview 3.4.1.

`target_updated` *`target ?time? ?debug-level?`*
: Marks the specified target as updated. Time is measures in seconds since 1970. If no *`time`* is specified, `[clock seconds]` is used. *`debug-level`* tells on which debugging level this operation shall be visible, and defaults to 1. 0 means that an update message appears even if no debugging options have been specified.

`target_untouched` *`target ?time? ?debug-level?`*
: Marks the specified target as unchanged, so it will only cause dependent targets to be updated if they are newer. *`time`* is again the timestamp in seconds. It defaults to the file modification time if the targets exists, and to 0 if not. *`debug-level`*: see `target_updated`.

`target_failed` *`target msg ?debug-level?`*
: Marks the specified target as failed, meaning that TMK was not able to generate it at all. This will cause dependent targets to fail, too. *`msg`* is a string telling the reaseon for the failure. *`debug-level`*: see `target_updated`.

`target_state` *`target ?varName?`*
: Returns the state of the specified target if it is already in the internal cache, or ”” if the target is not in the cache. If the result is `updated` or `untouched`, the stored timestamp is written into the variable `$varName`. If `failed` is returned, `$varName` will contain a message explaining the reason for the failure.

Commands 3.4.1: target state operations

Using one of these functions, you can at any time (e.g. while parsing the `TMakefile`, or after executing a rule) pretend that a certain target has just changed, has not been touched, or has failed to be built. This can for example be very useful if the up-to-date test for a certain target is a bit more complicated than just checking the time stamp. For example, let's say that some file `X` must be rebuilt whenever it does not exist or the the file `X.aux` is newer *and* contains the phrase `"need to rebuild X"`. This could be done as follows:

```
target X {X.aux} {
   if {[file exists $TARGET] && [file exists $SRC]} {
     if {![grep "need to rebuild $TARGET" $SRC]} {
       target_untouched $TARGET
       break
     }
   }
   puts "rebuilding $TARGET"
   ...
}
```

When the target script is executed, we know that either the target does not exist, or it is older that the `.aux` file. So we only need to check the case in which both source and target exist, and look if the desired string is contained in the file (which can be done by TMK's `grep` command, for example). If the string is not there, then the target does not need to be updated, and so the execution of the target script will not cause any dependent targets to be rebuilt.

Note that in order to cause a premature end of the target script, you can use the TCL `break` command to skip the rest of the rule.

3.5 Special/Pseudo Targets

Usually, target names represent files that are created by applying the corresponding rule. If the rule does not really generate that file, the target is called a *pseudo target*. For example, you can define a default target `help` that outputs the available "real" targets for the user:

```
target xy ... { ... }
target abc ... { ... }
target help ALWAYS_BUILD {
    puts stderr "available targets:"
    puts stderr "  xy:  ..."
    puts stderr "  abc: ..."
}
build help
```

Now, if the user just calls `tmk`, the pseudo target will simply output some informative text.

3.5.1 Target `ALWYAS_BUILD`

The target name `ALWAYS_BUILD` is simply a pseudo target that will never be up to date. Internally, TMK just marks this target as `updated` in its target cache (cf. the `target_updated` function in Sec. 3.4.5). So targets that somehow depend on `ALWAYS_BUILD` will have to be rebuilt every time.

In the above example, one could also specify an empty source file list ({}) instead of the pseudo target `ALWAYS_BUILD`. If a target has no primary dependencies, this means that it can be built *unconditionally*. This is nearly the same as if specifying `ALWAS_BUILD`, but an unconditional target does not need to be rebuilt if it already exists and does not have any further dependencies (cf. Sec. 3.3).

3.5.2 Target `clean`

Another predefined target is called *clean*. Its task is to delete all files that have been generated through running TMK. If you are using the target directory mechanism as described in Sec. 3.7, this is quite easy since TMK only needs to remove the directory that contains all the generated targets. If this mechanism is switched off, you should also define how to remove the results of your targets and modules. You can do this by simply defining new cleaning targets and make `clean` depend on them:

```
# these are the locally defined targets
target {a b c xyz} ...
target {bla blub} ...
# this is how to clean up
target myclean ALWAYS_BUILD {
     file delete -force -- a b c xyz bla blub
}
depend clean myclean
```

In the above example, the `myclean` target removes all potentially generated files. Since `clean` depends on it, `myclean` will also be executed when the user calls `tmk clean`.

3.5.3 Target `depend`

Another kind of predefined pseudo target is called `depend`. This is mainly used in conjunction with compilation (cf. Chapter 6), but in general its purpose is to update files that contain dependency information. For example, in the context of compilation, the compilers output information about which header and source files are included from within other files. The compilation modules store this information in *dependency files* (usually called `.dep`), and processes these files in order to generate secondary dependencies before starting the up-to-date check. If file or directory names are changed, it may sometimes be necessary to update the dependency files. This is done by calling `tmk depend`. As for the `clean` target, you should provide additional `depend` targets if you create and process dependency files other than the predefined ones.

The TMK default module defines a helper function named `read_dependencies`, which parses a dependency information file and returns the list of dependencies for the specified target. If the file does not exist, the function returns the value `ALWAYS_BUILD`, so that the target will be built unconditionally.

The file format of a dependency file is that produced by tools such as `makedepend` for use in a traditional `Makefile`. It simply contains lines of the form

```
    target-file:  ?file1?  ?file2?  ?file3?  ...
```

and long lines may be split over several lines by putting a backslash (”\”) at the end of the partial line. This representation translates directly into the TMK function call

```
    depend target-file {file1 file2 file3 ...}
```

3.5.4 Parameter-Changing Targets

You can also use pseudo targets to pass optional parameters to your `TMakefile` or script. For example, you may have a target that depends on some global variable:

```
set N 5
target hello {} {
    for {set i 0} {$i < $N} {incr i} {
        puts -nonewline "Hello!"
    }
}
```

This silly example prints its message a certain number of times, as specified by the global variable `N`. If you want to let the user change that number, you could specify another pseudo target which redefines `N`:

```
target n=* {} {
    set N $0
}
```

Now, if you specify an additional target like `n=10` on the command line, TMK will try to build that target, and set N accordingly:

```
    tmk n=3 hello
    -> Hello!Hello!Hello!
```

Here it is important to watch the order in which the targets are specified. TMK processes the targets in the order of their `build` commands, or in the order specified at the command line. If TMK would first build `hello` and then `n=...`, the variable `$N` would be set after it has been used.

3.6 Exceptions and Exclusions

When you use a number of general rules for building your projects, e.g. by means of *modules* as described later in Chapter 4, you often come to a point where you would like to define an exception from the usual rule, just for a certain target or a class of targets. This can either be done by temporarily changing the value of some variable that is used within the rule, or by defining a totally different rule just for these targets.

Another frequent operation is to exclude certain targets from being processed at all (e.g. a program that would not compile, but it currently not really needed), or to avoid the checking of time stamps for certain directories which are considered "constant" because it is known that these (system) files do not change often.

3.6.1 Changing Variables Within Certain Rules

The `exception` method provided by TMK offers a simple means for achieving all this. Have a look at the following example:

```
set FLAGS "x y z"
target {A B1 B2 C} {} {
    puts "$TARGET: FLAGS = $FLAGS"
}
build {A B1 B2 C}
```

The example is simple enough; it defines the same dummy rule for four different targets. The rule uses the global variable `$FLAGS`. Now we add some exceptions:

```
exception {B1} {FLAGS} {
    lremove FLAGS y
}
exception {B*} {FLAGS} {
    lappend FLAGS abc
}
```

The first exception command removes something from `FLAGS` specifically for target `B1`. The second exception appends something to `FLAGS` for all targets starting with the letter `B`.

The `exception` command takes three parameters: a list of target patterns for which the exception is going to be executed, a list of global variables which will be saved before the exception and restored afterwards, and the actual exception script that will be executed for the specified targets.

Now if you run TMK, you get the following output:

```
-> A: FLAGS = x y z
-> B1: FLAGS = x z abc
-> B2: FLAGS = x y z abc
-> C: FLAGS = x y z
```

As you see, you can define multiple execptions for the same target. However, the order in which the exception scripts are executed is not defined. TMK not only restores the specified variables to their previous values, but it also unsets the specified variables if they have not been set before executing the exception. If you change the value of some variable that you do not specify in the argument list, then this change will be globally visible for all commands that are executed lateron. However, you should never use this effect for globally modifying variables, since the side effect created by this kind of programming can become quite monstruous, especially in the context of rules which are executed in arbitrary order.

3.6.2 Replacing Rule Scripts

So far we have only changed the value of variables in order to change what happens – after this modifications the rule script has been executed as usual. It is also possible to override the rule script completely and define your own actions through the exception script.

```
target {A1 A2 A3} {} {
    puts "usual rule for $TARGET"
}
exception {A1} {} {
    puts "additional commands for $TARGET"
}
exception {A2} {} {
    puts "alternative rule for $TARGET"
    exception_return
}
build {A1 A2 A3}
```

In this example, a common rule for three targets is defined. For target A1, an exception provides an *additional* rule that is executed before the usual rule. For target A2 however, the special command exception_return has the effect that the usual rule will not be executed , so that the scripts provided by all matching exceptions will *replace* the usual rule. If you specify multiple exceptions for the same target and use exception_return in one or more exceptions, all exceptions will be executed in an unspecified order, and after that the rule will be skipped.

3.6.3 Excluding Targets and Dependencies

If you want to prevent some target or target pattern from being processed at all, simply add it to the list $EXCLUDE. Whenever TMK is told to build a target matching any of the $EXCLUDE glob-style patterns (cf. Sec. 2.4), the target will be ignored, and when it appears as primary or secondary dependency, it will be removed from that list of dependencies.

This behaviour alone can result in targets with an empty source file list, e.g. if you exclude some .c file and then TMK wants to execute the rule to generate the corresponding .o file. So TMK checks if due to exclusion the source file list (the primary dependencies only) has become empty. If so, the target is also excluded recursively. Here is an example:

```
target BC {B C} {puts "$TARGET <- $SRC"}
target {A B C} {$ROOT.src} {puts "$TARGET <- $SRC"}
target {A.src B.src C.src} {} {puts "$TARGET <- scratch"}

build {A BC}
lappend EXCLUDE B C.src
```

In this example, the targets B and C.src are excluded from the building process. The result is that neither C, nor BC can be built. The corresponding TMK output looks like this:

```
    A.src <- scratch
    A <- A.src
    tmk: BC skipped due to exclusion
```

The debugging output (using tmk -debug, cf. Sec. 3.8) is a bit more explicit about the exclusions that take place:

```
tmk: [dbg] toplevel targets: A BC
Linux2.2/A.src <- scratch
```

```
Linux2.2/A <- Linux2.2/A.src
tmk: [dbg] excluding [B] from BC’s primary dependencies
tmk: [dbg] BC <- [C], []
tmk: [dbg] excluding [C.src] from C’s primary dependencies
tmk: [dbg] C <- [], []
tmk: [dbg] skipping C due to exclusion (no source files left)
tmk: [dbg] skipping BC due to exclusion (no source files left)
tmk: BC skipped due to exclusion
```

For some files, especially secondary dependencies, it has turned out to be reasonable excluding these files immediately before they are considered in the depedency chain computation. TMK allows you to define exclusion patterns so that files matching any of these patterns will be discarded when passing them to the `depend` command. This completely avoids the timestamp check for these files, and can thus result in a noticable speedup.

For example, you know that some directory `/usr/bla` only contains system files which are nearly never updated. However, files from that directory are often included into your source files, resulting in a large number of secondary dependencies. In this case you write

```
lappend DEPEND_EXCLUDE /usr/bla/*
```

and the dependencies created by these files will be discarded as soon as they are passed to the `depend` function.

Please note that `$DEPEND_EXCLUDE` must be set *before* `depend` is called, since it has an immediate effect on the execution of the `depend` function. Furthermore, although you can use T-expressions (e.g. `$TARGET` or `$ROOT`) in the declaration of secondary dependencies (cf. Sec. 3.5.3), these will not yet be expanded when the dependency exclusion test is performed. This means that the following example does not work:

```
# this won’t work!
lappend DEPEND_EXCLUDE xy*.bar
depend *.foo {$ROOT.bar}
```

The above example will not exclude all dependencies of the form `xy*.bar`, since the unexpanded dependency file name in the `depend` call, `$ROOT.bar`, does not start with `xy`.

3.7 Target Directories

As already mentioned in some of the previous sections, TMK tries to place all generated targets into a special directory that depends on the architecture of the current machine. For example, if you generate a target named `filename` on a machine with the operating system *Linux* version 2.2, the `$TARGET` variable inside the rule will expand to `Linux2.2/filename`. If you are on a machine running IRIX, all targets will be generated in some directory like `IRIX6.5`.

This means that as long as you use the provided T-variables inside the rule scripts, TMK will put all generated files into an architecture-dependent directory for you. This is very convenient if you work on different machines or under different operating systems, especially if you develop software for multiple platforms. Since the directory name changes with the architecture, the generated code from mutliple systems will never get mixed.

```
target x {} {
    puts $TARGET
    puts [targetname_short $TARGET]
    puts [targetname_long  $TARGET]
}
build x
```

If you run the above example under Linux 2.2 machine, you should get the output `Linux2.2/x`, `x`, and `Linux2.2/x`.

As you see from the example, you can remove the architecture-specific part of a filename by using the procedure `targetname_short`. Its counterpart, `targetname_long`, includes the target directory if the architecture-dependent building is switched on.

For some applications, this default behaviour is not desired. You can switch off the architecture-dependent target location by setting the global variable `$USE_ARCH` to 0. In that case, the above example would yield three times the same result, which is simply `x`.

The name of the target directory is specified in the varibale `$ARCH`. The default value of `$ARCH` is set like this:

```
if { $CODELEVEL == "dbg" } {
    set_ifndef ARCH ${OS}${OSVER}
} else {
    set_ifndef ARCH ${OS}${OSVER}_${CODELEVEL}
}
```

`$OS` and `$OSVER` contain the name and version of the detected operating system, e.g. `Linux` and `2.2`, or `IRIX` and `6.5`. `$CODELEVEL` is a symbol for defining the "level" of generated code, e.g. the amount of debugging information or optimization when generating object files. You can find more information about code levels in Chapter 6. The default codelevel is `dbg` . You can query most of the configuration settings by invoking

```
    tmk -sysinfo
```

Usually, you do not modify the name of the target directory directly. If you do not want to use the default name for the target directory, you should change it in the project `TMakefile` (cf. Sec. 5), or even in the configuration of your TMK version [?]. The user should only influence the output directory through the code level command line options (e.g. `-prf` or `-max`). In certain cases, however, you may directly override the default value for `$ARCH` using the `-arch` command line option:

```
    tmk -arch Special
```

3.8 Debugging

Usually TMK only prints the explicitly specified log messages and the echoed system commands that are executed. However, you can request some additional information on what is actually going on through the `-debug` command line option. If you use `-debug` once, TMK will also print which files are read and the reasons why a target needs to be updated. Take this small example:

```
target {A B C D} {} {
  set txt "creating $TARGET"
  puts $txt
  write_file $TARGET txt
}
depend A {B C}
depend B {D}
build A
```

Now if you type `tmk -debug`, a lot more output will be produced. After some messages concerning the configuration, TMK outputs the actions that happend during the processing of the `TMakefile`, in a way similar to this:

```
tmk: [dbg] ----- begin processing TMakefile -----
tmk: [dbg] adding rule A <-
tmk: [dbg] adding rule B <-
tmk: [dbg] adding rule C <-
```

```
tmk: [dbg] adding rule D <-
tmk: [dbg] adding secondaries: A <- [B C]
tmk: [dbg] adding secondaries: B <- [D]
tmk: [dbg] adding default target A
tmk: [dbg] ----- end processing TMakefile -----
```

This output corresponds directly to what is written in the `TMakefile`. After having processed the `TMakefile`, the target building process starts with the toplevel targets. Here are some of the messages from the debugging output:

```
tmk: [dbg] toplevel targets: A
tmk: [dbg] A <- [B C]
tmk: [dbg] B <- [D]
tmk: [dbg] D <- []
tmk: [dbg] Linux2.2/D gets built because it has no
           prerequisites and does not exist
creating Linux2.2/D
tmk: [dbg] B must be built because Linux2.2/D has been updated
tmk: [dbg] B must be built because it does not exist.
creating Linux2.2/B
tmk: [dbg] A must be built because Linux2.2/B has been updated
tmk: [dbg] C <- []
tmk: [dbg] Linux2.2/C gets built because it has no
           prerequisites and does not exist
creating Linux2.2/C
tmk: [dbg] A must be built because Linux2.2/C has been updated
tmk: [dbg] A must be built because it does not exist.
creating Linux2.2/A
```

As you can see, you can easily follow the chain of events and conditions that TMK uses for processing its rules. If you get a lot more debugging output and you want to follow just the chain of reasoning, you can simply search for the word “because” in the output.

Now that all the targets have been created, let us pretend that the target `C` has just been freshly updated (and thus is newer than all other targets):

```
    tmk -debug -up C
```

Now the debugging output looks like this:

```
tmk: [dbg] marking C as updated (Thu Jan 01 01:00:00 "MET 1970)
tmk: [dbg] toplevel targets: A
tmk: [dbg] A <- [B C]
```

```
tmk: [dbg] B <- [D]
tmk: [dbg] D <- []
tmk: [dbg] nothing to be done for IRIX6.5/D
tmk: [dbg] nothing to be done for IRIX6.5/B
tmk: [dbg] IRIX6.5/A must be built because C has been updated
creating IRIX6.5/A
```

If for some reason you want to have even more debugging information, you can specify `-debug` twice or even more often. With increasing debugging level, more and more detailed information is printed.

If you want to see what will happen when you call TMK, but you do not dare to really execute it, use the `-pretend` command line option. In that case, the target rules will not be executed, but merely printed to the standard output. Unfortunately, this output does not always give satisfactory results, since the rules may still contain unexpanded variables and procedure calls, so that you may not only see the resulting primitive commands.

In addition to display all the debugging information, you may also selectively trace individual variables and procedures, e.g. if you want to know which value a variable has, or where it is modified. So if you want to know who modifies the `$qt::DIR` variable and the `$cxx::FLAGS` variable, you can do as follows:

```
    tmk -vtrace ::qt::DIR -vtrace ::cxx::FLAGS
```

Of course you can use these options along with all other operations of TMK, e.g. combine it with the `-debug` option, or follow the configuration system using `-reconfig`.

If you want to track procedure calls rather than variable modifications, you can use the `-ptrace` option in the same fashion. Just specify the fully qualified procedure name, and you will be shown all calls to that procedure, along with the corresponding arguments. Note, however, that procedure tracking only works if the procedure has already been defined either in the TMK core system, or before reading the `TMakefile`.

The `-rules` option lets you review all the rules that are defined by the used modules, the local `TMakefile`, the `TMakefile.proj` (cf. Sec. 5 etc.). Instead of building the targets, the complete active rule database is printed to the standard output. In the case of our example, the output for target A looks like this:

```
A <- :

  set txt "creating $TARGET"
  puts $txt
  write_file $TARGET txt
```

```
A secondary dependencies:
     B C
```

CHAPTER 4

Modules

Technically, modules are not much more than files containing TCL/TMK source code. Conceptually, a module represents a well-defined set of target definitions and parameters that can be used to generate a certain class of targets. This section explains how to use and write modules, rather than going into the details of individual modules. For more information about some of the provided modules, please have a look at Chapter 6 (compiling and linking), at the TMK reference manual [**?**]. You may find even more modules (e.g. some that are not included in the TMK distribution) on the TMK web pages [**?**].

4.1 Loading a Module

Modules are loaded explicitly from within the `TMakefile` using the `module` command, or via the command line using the `-mod` option. The `module` command in a `TMakefile` is excuted directly, whereas the command line option leads to the execution of the corresponding command after the parsing of the `TMakefile`. Here is an example `TMakefile`:

```
module {cxx qt}
```

The only argument to the `module` command is a list of modules, like `cxx` and `qt` in this case. The above example will cause the following action:

- TMK looks for certain *module macro variables* in the namespaces `cxx` and `qt` (cf. Sec. 2.11). Some of these variables will trigger a certain action, for example the variable `qt::DEPEND` defines which other modules are required so that the QT module can work (cf. Sec. 4.4)
- TMK searches the files `cxx.tmk` and `qt.tmk` (all lower case) in the following directories:
  - ./
  - all ":"-separated paths listed in the `TMK_MODULE_PATH` environment variable
  - the `modules` subdirectory of the TMK home directory (which is usually set via the environment variable `TMK_HOME`)

  The first file that is found will be read and executed in the corresponding module namespace.
- if for some module neither any module-specific variable nor a module file was found, TMK exits with an error message
- TMK appends the module names to the `$MODULES` variable; new modules are only loaded if they are not already contained in `$MODULES`

In addition to loading a module "just so", you can also require a specific *version* of a module. A version is represented by an arbitrary version number string, and you can request a specific version by appending a version string to the module name, separated by a double colon ("::") like for a namespace:

```
module {cxx}
module {glut::3.7 qt::2.0.1}
```

In the above example, the current version of the C++ module is requested, along with specific version of the modules `glut` and `qt` . Note that you cannot load module versions that have not been configured (cf. next section), and more importantly, you cannot load two different versions of the same module at a time. You can query the version of a loaded module via the variable `$`*`module-name`*`::VERSION`. If you want to find out which version of a module are installed on your TMK system, type

```
tmk -modver module-name
```

and TMK will output all available versions of the named module.

4.2 Delayed Evaluation and Variable Initialization

Since a module is loaded and executed immediately when the `module` command is read, it is important to think about the order and time of the operations executed by the module. The general idea is that although a `TMakefile` contains a *sequence* of commands, this should only define a number of rules and parameters in a *declarative* manner. This means that it should usually not matter whether a global variable is set before or after loading a module.

In order to support this idea, TMK provides the `eval_later` command. Its only argument, an arbitrary script, will be executed *after* the `TMakefile` has been parsed and executed completely. If you define multiple `eval_later` scripts, they will be executed in the order of their definition.

Usually, the actual module actions are all delayed until after the end of the `TMakefile`, or the module does only define a number of targets or dependencies anyway. Here is an example module:

```
# mymodule.tmk
set_ifndef USE_IT 1
set_ifndef FLAGS "1 2 3"
eval_later {
  if $USE_IT {
    ...
  }
}
```

There are two important things about this simple example. First, variables are assigned their *default values* in the module using the `set_ifndef` command (cf. Sec. 2.3). This ensures that if the user defines some of the values in a different way *before* loading the module, they will not be overridden by the default values.

Setting variables' values at the right time becomes even more difficult when the user desires to use and *modify* the default value. In that case, the user *must* make these modifications *after* loading the module, like in this example:

```
module mymodule
lappend mymodule::FLAGS 7 8 9
```

After executing this example, the value of `$mymodule::FLAGS` is `"1 2 3 7 8 9"`. If the `lappend` line had been executed *before* the module command, the result would have been `"7 8 9"`, since the `set_ifndef` command in the module would have had no effect.

4.3 Module Configuration

One of the general ideas of TMK is that the module file should only define the geneal rules how to deal with a certain task, and system-dependent things like package locations, executable names, system-dependent flags etc. will be put into a *config file*.

To this end, not all variables should be set directly in the module file, but rather in the appropriate *architecture* and *site* config files. A lot can be said about the general organization of TMK's configuration system [**?**], but here we only briefly sketch the most important facts from the user's point of view.

In general, a module serves as an interface to a certain software package, as for example a library, a converter tool, or a set of compilers. Usually the module needs to know where this package is installed on the current system. Therefore, TMK processes so-called *site-config* files that can be found in the subdirectory `config/site-config` of the TMK installation directory. After processing some general and architecture-dependent configuration files, TMK looks for site-specific files in the site config directory, in the following order:

- `site-config.tmk`
- `${DOMAIN}`
- `${DOMAIN}:${OSCLASS}`
- `${HOST}:${DOMAIN}`
- `${HOST}:${DOMAIN}:${OSCLASS}`

where `$HOST` is the name of the current machine, `$DOMAIN` the name of the network domain or `localdomain`, and `$OSCLASS` the current operating system's type, e.g. `unix`, `windows`, or `macintosh`. Each of these files is first searched in the subdirectory `config/site` of TMK's installation location, and directly afterwards in the user's private site config directory, e.g. `<home dir>/.tmk/site-config` on UNIX-like systems. This way, the user can override all site-config settings by settings in his or her home directory.

You can make TMK output the values of the system-dependent variables, and also output the order of config files that are being read. Try the following two commands:

```
tmk -sysinfo
tmk -reconfig
```

TMK only performs the "file walk" across the configuration tree when it is started for the first time on a machine or when `tmk -reconfig` is called. It stores the resulting configuration values in a *cache file*. This cache file has a unique name for

each operating system on each host. TMK outputs this cache file name along with other information upon using the `-reconfig` option.
In order to define such cached variables, the config files use the `config` command provided by TMK. `config` has a number of sub-commands and can be used to manipulate, query, read, and save the config cache. From the user's point of view, the most important subcommands are `variable`, `set`, and `proc`, since they provide the means to define new variables and procedures and add the to the cache. After being defined, they can be used as any other TCL/TMK variables/procedures, the only difference is that they will be saved in the config cache lateron. Here is an example from an existing site-config file:

```
  namespace eval qt {
  config set DEPEND {gui}
  config set LIBPATH /opt/pckg/qt2/lib32
  config set INCPATH /opt/pckg/qt2/include
  config set LIBS   {qt}
}
namespace eval qgl {
  config set DEPEND {qt opengl}
  config set LIBS   {qgl}
}
```

These commands define some variables in the namespaces of the `qt` and `qgl` modules, so afterwards there exist TCL variables like `qt::DIR` or `qgl::LIBS`. Instead of the above syntax, you can also use:

```
config set qt::DEPEND  {gui}
config set qt::LIBPATH /opt/pckg/qt2/lib32
config set qt::INCPATH /opt/pckg/qt2/include
config set qt::LIBS    {qt}
config set qgl::DEPEND {qt opengl}
config set qgl::LIBS   {qgl}
```

In contrast to the TCL `set` command, `config set`lso creates the necessary namespace if it does not already exist. So you can use any kind of qualified or unqualified names in conjunction with `config set`. In the same manner, you can define cached config procedures with the `config proc` command.

If you want to configure different *versions* of the same module, just add children namespaces to the module's namespace, and redefine variables and procedures there. If a specific module version is requested, TMK will recurse down into the right children namespace and import all variables and procedures from there into the module's namespace:

```
namespace eval qt {
  config set DEPEND {gui}
  config set LIBPATH /opt/pckg/qt2/lib32
  config set INCPATH /opt/pckg/qt2/include
  config set LIBS   {qt}

  namespace eval 2.0.1 {
    config set LIBPATH /opt/pckg/qt-2.0.1/lib32
    config set INCPATH /opt/pckg/qt-2.0.1/include
  }
}
```

For this example, the default version of `qt` will be found in `/opt/pckg/qt2`, which may for example be a link to the latest version, whereas module `qt::2.0.1` will set up the QT version in directory `/opt/pckg/qt-2.0.1`. Note that you only need to override those variables that differ from the parent version.

4.4 Module Macro Variables

As already briefly mentioned in Sec. 4.1, there are a number of variables that trigger certain associated actions when the corresponding module is executed, even if there is no module file at all. This is very convenient for very simple "modules", like for example those that merely define a number of system-dependent library names and include paths. E.g. if the command `module xyz` is excuted, TMK will look for some special variables in the namespace `xyz`, and trigger actions for the following ones:

- `$xyz::DEPEND`: contains a list of modules on which `xyz` depends. They will be executed before the `xyz` module file is actually read.
- `$xyz::LDFLAGS`: append the specified flags to `$link::FLAGS` (see Chap. 6)
- `$xyz::LIBPATH`: appends the specified library directories to `$link::LIBPATH`
- `$xyz::INCPATH`: add compiler flags to search header files in the specified include directories (e.g. `-I...`)
- `$XYZ::LIBS`: append the specified libraries to `$link::SYSLIBS`

The definition of the predefined module macro variables can be found in the source file `module_macro_vars.tmk` in the `src` subdirectory of the TMK installation. If needed, you can simply add more such definitions via the config files.

4.5 Example: a LaTeXModule

. . . to appear . . .

CHAPTER 5

Projects And Directories

So far in this manual, we have only considered small examples that exist on their own and are implemented in a single `TMakefile` in some directory. When creating larger projects, e.g. in the context of software development, web page management, or distributed systems administration, it is more likely that you organize tasks in a hierarchical manner, most often using a *directory tree*. In this chapter, you learn how TMK supports the project concept, and how you can effectively use TMK to organize your own projects.

5.1 Project Directory and Project Files

As already mentioned, projects are usually structured as a directory tree. In each subdirectory, there is a `TMakefile` that defines the *local* tasks that have to be processed there. Before that, TMK searches for a *project file* in the root of the hierarchy. This file, called `TMakefile.proj`, contains a number of definitions that are *globally* valid in the complete project.

So when you start TMK in some directory, it actually first look for a file called `TMakefile.proj` in the current directory. If it cannot find it, TMK recursively looks in the parent directory until it reaches the root of you directory tree. If TMK finds a project file, it sets the variable `$PROJDIR` to the corresponding directory, and

`$PROJROOT` to the parent of that directory. The project `TMakefile` may not be located in the root directory, or unsatisfactory results will occur. If TMK does not find a project file, `$PROJDIR` is set to the current directory, and `$PROJROOT` to its parent.

The two variables `$PROJDIR` and `$PROJROOT` can be used conveniently for addressing files in other parts of the project without using the absolute location of the project directory. Additionally, TMK defines the variables `$SUBDIR` and `$DIRTAIL` that contain the subdirectory relative to the project root, and the local name (the last path segment) of the current subdirectory. For example, if we were in `/x/a/b/c`, and if the `TMakefile.proj` is in `/x/a`, then the following values would be set:

```
$PROJROOT = /x
$PROJDIR  = /x/a
$SUBDIR   = a/b/c
$DIRTAIL  = c
```

The `TMakefile.proj` usually contains a number of project-global definitions shared by all users of the project. If a user wants to modify one or more is these definitions for only himself or herself, this should not be done in the common project file (especially if the file is version-controlled and the user would risk to commit his or her private changes by accident). So there is another file that TMK looks for in the same place as the `TMakefile.proj`, which is called `TMakefile.priv`. It is read immediately after the `TMakefile.proj`, and can override or change all the previous definitions.

If for some reason you want to specify the location of the `TMakefile.proj` or `TMakefile.priv` explicitly, you can do this using the TMK command line options `-proj` and `-priv` , or prevent TMK from reading the corresponding file by specifying `-noproj` or `-nopriv` . A `TMakefile.priv` is only read after a `TMakefile.proj` has been read. See `tmk -help` for details.

5.2 Subdirectory Processing

As already mentioned, a project usually consists of a directory tree with a global project file and a local control file for each subdirectory. Since the users often desire some simple means to build the complete project through only a single command, TMK supports recursive subdirectory processing. The general idea is that all the subdirectories are built *before* the current directory is processed (a so-called bottom-up approach)[1]. The corresponding TMK command is called `subdir`, and it immediately

[1] In future TMK versions, it may be possible to toggle this behaviour, so that parent directory targets are built before those in their subdirectories.

calls TMK recursivly for all the specified directories, e.g.

```
subdir [glob *]
```

In the above example, the `glob` command will return a list of all files in the current directory. TMK discards all arguments that are not directories, and it only processes directories that contain a `TMakefile`.

In most projects however, it is usually more convenient to list subdirectories explicitly in a reasonalble order, since often one directory depends on the other one. Here is an example:

```
subdir {base base_apps extensions extensions_apps}
```

The names in the example suggest to structure the directory tree in such way that the applications (e.g. the executables in the `..._apps` directories) are separated from the actual functionality that you might want to put into a reusable library. This has also the advantage that you have different `TMakefile`'s for applications and libraries, which often makes things a lot easier.

By default, TMK executes all the `subdir` commands it finds in the `TMakefile` immediately. You can prevent TMK from recursing into subdirectories using the `-local` command line option.

For every remaining subdirectory, a new child process with another instance of TMK is started. The executed command is basically

```
cmd $TMK $ARGS
```

and TMK will add some more flags for specifying the location of the `TMakefile.proj` and `TMakefile.priv` as well as the output prefix text for the subprocess (`-prefix` option). The `$TMK` variable holds the name of the TCL shell (usually called `tclsh`) and the name of the TMK script that the user has invoked. `$ARGS` contains all the command line options that have been used for the invoking TMK process, except the makefile location and prefix flags mentioned before.

If you want to track the TMK subprocess calls, please use the `-debug` command line option, where the actually executed commands will be shown.

When TMK recurses into a subdirectory and there is a file called `TMakefile.proj` in that subdirectory, then this file is used as new project file, and `$PROJDIR` etc. are set accordingly. This means that in special cases you may have something like *sub-projects* in your directory structures, or that you can place arbitary `TMakefile`'s in the parent directory of your projects that will build all the projects in a specified order with just one call to TMK. Be careful, however, since sub-projects are indeed projects on their own, so variables like `$SUBDIR` will not reflect the fact that it is a part of a larger project. However, you can include the definitions of the parent project file manually (by using TCL's `source` command), and you could even change the variables `$PROJROOT`, `$PROJDIR`, and `$SUBDIR` in the sub-`TMakefile.proj` to reflect the sub-project structure that you intend.

5.3 Project Location Path

As already mentioned in the previous sections, one general idea of TMK's project concept is that all project-related file and directory names are specified in the same location-independent manner as in the `$SUBDIR` variable, meaning that a project-relative path starts with the (directory) name of the project, followed by the actual subdirectory within that project. Or, in other words, the project-relative path is just the path from the parent of the project directory (`$PROJROOT`) down to the actual file or directory. If this notation is used consequently everywhere, it is really easy to relocate projects and yet maintain a unique naming scheme throughout all projects, given, that no project name is used twice.

In order to facilitate transparent project relocation, TMK defines the `$PROJ_LOCATIONS` list and a the `find_proj_file` function in the `default` module (which is always loaded). The idea is that whenever you need to find a file that has been specified by a project-relative path, `find_proj_file` will find the absolute location of that file for you.

To that end, the user must include the parent directories of all projects in the `$PROJ_LOCATIONS` list. `find_proj_file` will always look in `$PROJROOT` (the parent of the current project) first, and then in all the directories specified in `$PROJ_LOCATIONS`.

If you want to see an example of how this mechanism can be used, have a look at the way the `link` module handles project libraries and include paths (Sec. 6.3).

5.4 An Example Project

Currently, we do not provide a very sophisticated example project here. Please have a look at the `demo-proj` directory in the TMK dirstibution, or on the TMK web pages.

CHAPTER 6

Compiling and Linking

This chapter teaches you how TMK can be used for software development. This involves things such as compiling source code into object files and linking object files into executables and libraries, but it also includes other tasks such as project organization and distribution building.

The first section gives a quick start for those who just want to compile a program using TMK. After that, you can learn more about some advanced parameters of the compilation and linking modules, followed by some general ideas about the structuring of larger code projects and the automatic creation of executable distributions.

Finally, Section 6.7 briefly describes some related modules, such as for different code generators, libraries, and versioning systems.

6.1 C++ Compiling and Linking with TMK

If you want to compile and link a program using TMK, you should first create directory containing your source files. Let us assume that you have the source files `a.cxx`, `b.cxx`, and `myprog.cxx`, and that you would like to compile them all and make an executable called `myprog` (i.e. `myprog.cxx` contains the `main` routine for your executable). In that case, you create a `TMakefile` with just the following contents:

```
module cxx
```

This line tells TMK to load the module for compiling C++ code. Since at this point we do not need any non-standard options, TMK does not need any more information. When you call TMK in this directory, the following things will happen:

- All `.cxx` files in the directory will be compiled. The resulting object files will be named `.o` or `.obj`, depending on the operating system.
- All object files except for `myprog.o` will be put in a *local library* that will be named after the directory it is in.
- The executable will be produced by linking `myprog.o` with the library that contains the other object files. The suffix of the executable file is also system-dependent, for example it will be just `myprog` on UNIX systems, and `my-rog.exe` on Windows-style systems.

If you want to generate multiple executable files in the same directory, this works in exactly the same way, without any further TMK code. TMK will put all object files that do not correspond to executables in the library, and link all executables with that library. If no library object files are there, TMK just skips the library linking step.

6.1.1 Code Levels and Output Directory

All files generated by TMK are usually written into an architecture-dependent target directory (cf. 3.7), so that you can easily distinguish generated files from source files. The default name of that output directory consists of the operating system name and version, plus a suffix indicating the *code level* of the generated files. TMK knows the following code levels:

- `prf`: include as much profiling and debugging information in the generated code as possible, and perform no optimizations
- `dbg`: include debugging information in the generated code, and perform no optimizations
- `std`: standard optimizations and debugging information
- `opt`: higher-level optimizations (not CPU-dependent), no debugging information
- `max`: highest-level (e.g. CPU-dependent) optimizations, and no non-vital information in the generated code

You can choose between these code levels using the corresponding TMK command line options `-prf` , `-dbg` , `-std` , `-opt` , `-max` . The default code level is `dbg`. The current code level is stored in `$CODELEVEL`.

Since the platform as well as the code level are reflected in the output directory name, you can compile your software on different platforms and with different sets of flags without the chance that different code types will be mixed in the same directory. Furthermore, you can create further output directory and flag types on your own, by simply modifying the `$ARCH` variable (the output directory name) depending on certain flags. If you also use different C/C++ compilers on the same platform, you can include the following code in your `TMakefile`, or better in your `TMakefile.proj` (cf. Sec. 6.3.2):

```
# choose c and c++ compiler
ifdef CCOMP {
  set c::COMPILER   $CCOMP
  set cxx::COMPILER $CCOMP
  set ARCH $ARCH_$CCOMP
}
```

This small script will set the C and C++ compilers as well as the name of the output directory depending on the value of `$CCOMP`. You can now set `$CCOMP` somewhere in your `TMakefile`'s, or specify the compiler as a command line argument

```
tmk -D CCOMP=gcc ...
```

If the compiler and linker flags that are generated for each code level are not what you would like them to be, you can either change them in the compiler configuration [**?**], or you can manually modify the compiler's default flag variables, which are called

> *`language`*`::`*`compiler`*`::FLAGS_`*`codelevel`*
> and
> *`language`*`::`*`compiler`*`::LDFLAGS_`*`codelevel`*

where *language* is the language module name (e.g. `c` or `cxx`), *compiler* the compiler package name (e.g. `gcc` or `mipspro`), and *codelevel* is the codelevel (all in uppercase letters) for which to change the default flags. The `$LDFLAGS_...` variable is used for linking commands, whereas the simple `$FLAGS_...` flags are used for all other tasks. Here is a configuration example, althrough we strongly recommend to use the configuration system for these kinds of things:

```
# custom default flags for MIPSpro C++
namespace eval cxx::mipspro {
  set FLAGS_PRF "-g3"
  set FLAGS_DBG "-g"
  set FLAGS_STD "-g -O"
  set FLAGS_OPT "-OPT:Olimit=30000 -O2"
  set FLAGS_MAX "-OPT:Olimit=50000 -O3 -INLINE -LNO -
IPA"

  set LDFLAGS_PRF ""
  set LDFLAGS_DBG ""
  set LDFLAGS_STD ""
  set LDFLAGS_OPT ""
  set LDFLAGS_MAX "-IPA"
}
```

6.1.2 Linking External Libraries

TMK lets you specify *external libraries* (e.g. some math or GUI library) that are needed to run your programs or use your library. There are two different cases:

- you use functions or variables from the external library in the code that is put into the local library of the current directory
- you only use the external library within the code of an executable, but not within the local library

In both cases, you need to specify the external library when linking a local executable. In the first case, however, you want to make sure that whenever you use the local library fro a different place, the external library will also be used when generating executables. TMK does this by storing library information in so-called *libspec info files* (`.libspec`), and by reading these files when a project library is used (this is called *transitive linking*, cf. Sec. 6.3.2 and Sec. **??**).

In order to distinguish the two cases, TMK defines the following variables:

`$link::LIBPATH`: list of directories in which external libraries will be searched, in addition to the system's library search path

`$link::SYSLIBS`: short names of the libraries that are to be linked

`$link::EXE_LIBPATH`: like `$LIBPATH`, but only for linking the executable, not for using the local lib

`$link::EXE_SYSLIBS`: like `$SYSLIBS`, but only for linking the executable, not for using the local lib

---

Here is an example that adds the math library "m" as well as some non-standard library called `somelib` that can be found in the directory `/somewhere/lib`:

```
lappend link::SYSLIBS m somelib
lappend link::LIBPATH /somewhere/lib
```

The `link` module will add the corresponding linker options to the arguments that the compiler/linker program is called with.

Instead of manipulating the library variables manually, you should rather let *modules* do that job for you. In that case, the usually system-dependent library and path names would be put in a config file (cf. Sec. 4.3), and you would just call:

```
module {math somelib}
```

The above example affects both the local library and the local executables. If you want to restrict the effect of a library module to just the executables, you can do as follows:

```
link::exe_only {
  module {math somelib}
}
```

The `link::exe_only` command simply 'redirects' all read/write access to the `$LIBPATH`, `$link::SYSLIBS`, `$link::PROJLIBS`, and`$link::OPTIONS` variables (see also Sec. 6.3.2) so that they affect their corresponding `$EXE_...` counterparts. You can place arbitrary TCL/TMK code within the braces following the command.

6.1.3 Compiler/Linker Flags and Options

As we have seen before, modules and other commands usually generate compiler/linker flags that are appropriate to do what is desired. If, however, you want to add or modify some options of your own, you can do this by using `$c::FLAGS`, `$cxx::FLAGS`, and `$link::FLAGS`. The following example passes the two options "-DHELLO" and "-woff 15" to the compiler:

```
lappend cxx::FLAGS -DHELLO -woff 15
```

Since most of the time these kinds of flags are only valid for certain compilers, each compiler additionally has its own `$FLAGS` variable that can be set separately. Finally, both the `$cxx::FLAGS` and the compiler-specific flags will be used. For example, for the C++ compiler of the compiler system called `gcc` you could write:

```
lappend cxx::gcc::FLAGS -g
```

You can determine or modify which compilers are available and which compiler is chosen via the language's `$COMPILERS` and `$COMPILER` variable, e.g.:

```
tmk -f -
-> tmk: now entering tmk's interactive mode...
module cxx
puts $cxx::COMPILERS
-> gcc mipspro
puts $cxx::COMPILER
-> gcc
```

The same compiler environment may be suitable for multiple languages. For example, the gcc compiler package can also generate object files from C and Objective C code. For each language/compiler combination, there is a separate namespace with its complete set of variables, e.g. `c::gcc::FLAGS` and `cxx::gcc::FLAGS`.

6.2 Exclusions and Non-Standard Tasks

Whenever you have source files in your directory that you do not want to be compiled, or other files that you somehow do not want to be processed by TMK, simply add these files to the `$EXCLUDE` list:

```
lappend EXCLUDE old-prog.cxx
```

This will prevent TMK from using the specified file for generating any targets, so in this case no `old-prog.o` file will be generated.

TMK uses very simple heuristics for determining which source and object files will be linked into an executable. Basically, the current TMK implementation looks if there is a keyword resembling `... main(...)` in your source code file, and even does not recognize comments or preprocessor directives[1]. If due to some reason

[1]In future, this could be replaced by a language-independent implementation that looks into the object files and determines if there is any startup code in the object file.

TMK does not detect correctly which file should be processed as executables, you need to switch off the language's automatic executable detection, and list these files manually, e.g.:

```
module cxx

# switch off detection of executables for C++
set cxx::DETECT_EXE 0

# myprog1.o should generate an exe file
lappend link::PROGRAMS myprog1
```

This mechanism of explicitly specifying executables has another advantage: you can choose executable names that are not derived automatically from the object file names. You can do this by specifying a pair (*exe-name*, *obj-name*) instead of a single exe-name element:

```
set cxx::DETECT_EXE 0
# generate myApp instead of myprog1 from myprog1.o
lappend link::PROGRAMS {myApp myprog1.o}
```

6.3 Project Structure and Libraries

If you have more than just a few source and object files, you usually call it a *poject* and try to organize it in some consistent way, so that the project is split into several parts, each representing a logical sub-unit of the whole thing. As explained in more detail in Chapter 5, TMK supports this idea through the use of a project directory tree and recursive directory processing. Furthermore, the linker module provides some very convenient functions for plugging together code from different parts of a project.

6.3.1 Directory Structure and Recursive Processing

In order to build up a whole tree of directories for your project, you simply create all these directories and put one `TMakefile` in each. Then, for all directories that have subdirectories to be processed by TMK, you add a line like this:

```
subdir {subdir1 subdir2 subdir3 ...}
```

or, if the order in which the directories are processed does not matter, simply put

```
subdir [glob -nocomplain *]
```

which will simply process all directories that contain a `TMakefile` in arbitrary order.

Whenever you call TMK in one of the directories, it will first descend into all subdirectories and call TMK recursively there. After having done that, it will go back to the original directory and build all targets there. This order of operations is also called *bottom-up*, and you have to consider this strategy when building up your software tree. For efficiency reasons, in one directory you should not depend on code that is build *lateron* in another directory.

Within a single directory, TMK checks which targets depend on other targets, and builds the targets in the proper order automatically. But between *different* directories, the current TMK version does no recursive call to build things in the other directory when they are needed. So for now you have to rely on the fixed bottom-up building order, which is also the most efficient method considering the number of dependency checks that needs to be performed[2].

If you want to make sure that TMK only compiles in the current directory, but not in the subdirectories, you can switch off the recursive processing by typing

```
    tmk -local ...
```

6.3.2 Project Libraries

As already explained before, the default behaviour of the language and linker modules generates a local library in each subdirectory of a project. These are called *project libraries*. Let's assume that you are in subdirectory `a/b/c` of project `X`, and you want to use the library of subdir `x/y` from project Y. Then you simply specify

```
# we're in X/a/b/c, and link Y/x/y
lappend link::PROJLIBS Y/x/y
```

This way, you do not need to copy libs to special locations or care about the actual library names or paths. TMK assigns the project libraries unique names that are derived from the project name and project subdirectory path. So library `a` in project `X` won't be confused by the linker with the same library in project `Y`.

If you specify libraries to be linked via `$link::PROJLIBS` as in the example above (or via `$link::SYSLIBS`, respectively), TMK assumes that those libraries

[2]However, it is planned to support automatic inter-directory dependencies and recursive directory processing in a future release as an option.

are needed in conjunction with the local library as well as with the local executables. In contrast to that, you can specify that certain libraries are only needed for the executables, but not for the local library. As already explained in Sec. 6.1.2, TMK defines the variables `$link::EXE_PROJLIBS`, `$link::EXE_SYSLIBS`, and `$link::EXE_LIBPATH` as well as the `exe_only` command to support this.

6.3.3 Transitive Linking

idxlinking!transitive

Although the distinction described in the previous section is not important when only looking at one directory, it is very important for *transitive linking*, which is the recursive process to account for all libraries that are *prerequisites* for the libraries that are linked with the current targets. Let us consider the following project structure:

```
myproj/a
  link::SYSLIBS = m
myproj/b
  link::PROJLIBS = myproj/a
myproj/c
  link::SYSLIBS  = qt
  link::PROJLIBS = myproj/b
```

The example just lists which libraries are specified in which project subdirectories. Subdirectory `b/` uses code from `a/`, and `c/` uses code from `b`. For linking in directory `c/`, in the times of shared libraries and templates it is sometimes necessary to access not only the libraries specified there, but also all the prerequisites such as `myproj/a` and `m`. TMK's `link` module provides two ways of achieving this: the the meta linker options `transitive` and `lib_in_lib`. These options can be added to, and removed from the `$link::OPTIONS` variable, e.g.:

```
lappend ::link::OPTIONS "transitive"
lremove ::link::OPTIONS "lib_in_lib"
```

Adding the `lib_in_lib` switch will cause all libraries specified in `$link::SYSLIBS` and `$link:::PROJLIBS` to be linked into the local library, and should only be used with shared libraries. Since now the library has a reference to its prerequisites, transitive linking should be done automatically by any good linker without further work. However, this method has several drawbacks. First, not every linker is capable of transitive linking, and second the method creates a lot of references. If at some point you want to replace a shared library by a different version in a different location, the old references will persist until you rebuild all dependent libraries.

Therefore, TMK also offers a custom solution via the `transitive` option. When switching this on, TMK simply records all prerequsite library information (paths and libs) in so-called *libspec information files* for each local library. Whenever you use TMK to link such a library and the `transitive` option is switched on, the libspec information is added to the current set of libraries and paths, and so transitive linking is performed explicitly by TMK.

In our example on page 71, this would mean that a `.libspec` file `myproj/a` would record the need for the math library (`m`), and in `myproj/b` there would be a `.libspec` file pointing to `myproj/a` and its prerequisites, represented again by the math library `m`. Upon linking in directory `myproj/c`, TMK wants to link the lib in `myproj/b`, and so it additionally reads the `.libspec` file and therefore adds `myproj/a` and `m` to its current library specification, which is about what is needed. So far, this explicit transitive linking approach has proved to be very robust and convenient.

6.3.4 Project Locations

Another very nice feature of TMK's project management is that not all projects or parts of the project need to reside in the same location. For example, those projects that are not changed very frequently but used by a large number of people may reside in a central location that is provided to all users via a local area network. In this case, TMK should first look if a user has his or her own version of the project or subdirectory, and if not, use the centrally provided version.

In order to do this, you just need to specify the project location path via the `$PROJ_LOCATIONS` variable, e.g. in your `TMakefile.proj`.

```
lappend PROJ_LOCATIONS /net/current_projects
lappend PROJ_LOCATIONS /net/external_projects
```

If you do so, TMK will add all these directories to the include path of the compilers (e.g. by using something like `-I/net/current_projects`), and it will automatically search project libraries in these other locations.

For example, if you have a large shared project `XYZ`, and you only want to change a tiny bit in some subdirectory `XYZ/a/b/c`, then you only need to put this sudirectory and its parent directories in your personal working area. TMK and the compiler will search all include files and libraries in your personal directory first, and then in the `$PROJ_LOCATIONS` path.

6.3.5 Project-relative Path Specification in Source Files

In your code, you must specify include files relative to the parent of the project directories, much in the same way as you specify project libraries. Here is an example:

```
// this is a piece of C / C++ code
#include <OtherProj/a/b/c/some_definitions.h>
#include <ThisProj/proj_header.h>
```

Like this, the compiler will always use the first matching header file that it finds in the project locations path.

Please have a look at the `demo-proj` directory in the TMK distribution in order to see a practical example of how this project mechanism works.

6.4 Advanced Options

Of course you might not be satisfied with the default action of the compiler and linker modules. In that case, you can change their behaviour through a number of additional variables. Furthermore, you can switch off the automatic mechanism completely and specify the things to be generated by hand. The compiler and linker modules provide a number of functions and variables for making this easier. We cannot list all the possibilities here in this tutorial, but we list a few of them.

***language*::DETECT_OBJ** switch on/off automatic generation of object files. If set to 1, the language module will detect source files, and generate targets for the corresponding object files by calling the function `language::make_obj`. Default is 1.

***language*::DETECT_EXE** switch on/off automatic generation of executable files. If set to 1, the language module will derive from the source files whether the generated object file should be used to make an executable, and append the file name to the `$link::PROGRAMS` variable. Default is 1 for modules `c` and `cxx`.

link::MAKE_LIB switches the generation of a *static* library in each directory. The library contains all object files that are listed in `$link::LIB_OBJ`, but not in `$link::PROGRAMS`. The default setting is 0 on UNIX systems and currently 1 on Windows systems.

link::MAKE_SHLIB switches the generation of a shared library. See `$link::MAKE_LIB`.. Default value is 1 on UNIX systems and currently 0 on Windows systems.

link::MAKE_EXE switches the generation of executables. If on, TMK will generate targets for all the executable listed in `$link::PROGRAMS`. Default is 1.

In addition to these flags, TMK knows a number of so-called *meta compiler options*. These options are defined on an abstract level, and then implemented for each compiler in the best possible way. The meta options can be passed to the compilation and linking stage of the compiler separately, be setting the variables `$`*`language`*`::OPTIONS` and `$link::OPTIONS`. Furthermore, additional options can be specified for static or shared lib generation or executable generation by using `$link::LIB_OPTIONS`, `$link::SHLIB_OPTIONS`, and `$link::EXE_OPTIONS`.

The meta compiler/linker options have just been introduced to TMK. Usually compilers ignore meta options that they do not implement or that only make sense for a different operation stage. Currently there are the following options:

- `"transitive"`: tells the linker to also link libs that are prerequisites to the project libs that are currently used and specified. This is done by storing additional information in `.libspec` files along with each generated library. See Section 6.3 for an example.
- `"lib_in_lib"`: as an alternative to the `"transitive"` option, this options tells the linker to link prerequisite libraries into shared libraries. This is not a very nice way of solving the problem, since many different references to the same library will be created. See Section 6.3 for an example.
- `"circular"`: tells the linker to try to resolve circular inter-library dependencies. If a linker does not support this option, tmk usually simulates the effect by specifying each library twice instead of once (this nearly always works!)
- `"rpath"`: tells the linker to store runtime library path information with executables. This is needed for finding shared libraries at runtime.
- `"stl"`: tells the compiler to do everything that is necessary in order to support the C++ language standard template library. For example, MIPSpro compilers require the `-LANG:std` flag.

The currently used default meta options for all stages are {`transitive circular rpath`}. The best place for changing this behaviour is probably the `TMakefile.proj` of your project, e.g.:

```
# default meta compiler options
lappend link::OPTIONS transitive circular
lremove link::OPTIONS lib_in_lib
lremove link::OPTIONS stl
lremove cxx::OPTIONS  stl
lremove c::OPTIONS    stl
```

There are many more ways of manipulating and using the language and compiler modules, but they go beyond the scope of this tutorial. Please refer to the TMK reference manual and the "compiling and linking guide" that will appear soon.

6.5 Wrapping Libraries In Modules

If you want to write your `TMakefile`'s in a portable and configuration-independent way, you should not specifiy library paths and names explicitly, but rather wrap each library in a simple *module* as explained in Sec. 4. So for example, you can define a module named `math` that sets the appropriate paths and libs for using the math functions. A typical `TMakefile` would then rather look like this:

```
module {cxx math pthreads qt}
lappend PROJLIBS ...
```

Note that usually you should load the library modules *after* the language module, since the library paths may depend on the atual linker that is used on some systems such as Windows.

The advantages of the modules approach should be obvious, especially if you work on multiple platforms, or if library locations may change from time to time. In that case you only need to change the central configutation file rather than each `TMakefile` in order to migrate to the new software location.

However, there is a small design flaw in the current version of TMK, since automatic modules only add libraries and paths to the general variables `$link::LIBPATH` and `$link::SYSLIBS`, and you cannot choose to modify the corresponding `$link::EXE_...` variables instead. However, this is not really a serious problem since firstly, in a large project applications / test programs and software libraries should be placed in separate directories, and secondly, if really necessary you can move the libraries and paths from one variable to the other like this:

```
module qt
lremove link::LIBS    $qt::LIBS
lremove link::LIBPATH $qt::LIBPATH
set link::EXE_LIBS    [concat $link::EXE_LIBS    $qt::LIBS]
set link::EXE_LIBPATH [concat $link::EXE_LIBPATH $qt::LIBPATH]
```

Admittedly, this is not nice, but at least it is still portable...

Another important feature of modules is the version support. This means that you can configure multiple versions of the same module, and choose a specific version for those applications that need it. In general, you should try to use version-less module specifications wherever possible, and only use versions for software that really needs a specific (e.g. older) version of another package. Here is an example (cf. Sec. 4.1):

```
module {cxx}
module {glut::3.7 qt::2.0.1}
```

This example loads the generic C++ module, plus specific version of both the `glut` and `qt` modules. Note, however, that you cannot mix different versions of the same module in one `TMakefile`.

6.6 Binary Distributions

Binary distributions are a collection of executables and all shared libraries that are needed to run them. Furthermore, a distribution may contain arbitrary additional files, e.g. documentation, examples, or data. TMK defines a `dist` module to facilitate the generation of distributions. Especially, it finds all necessary libraries for the specified targets, and also creates a wrapper script for each executable that can be used to redefine the runtime library path or other environment variables that depend on the actual location of the distribution. The script determines its location automatically, so that the distribution can be relocated to any place without any problem.

The current TMK version only supports wrapper scripts for UNIX systems. Here is a simple example for how to create a distribution in a separate directory:

```
module dist
lappend dist::TARGETS projA/x/y/$ARCH/myexecutable
lappend dist::TARGETS projA/x/$ARCH/libprojA_x.so
```

This simple `TMakefile` causes the following action:

- TMK creates subdirectories `bin/` and `lib/` below the current output directory

$ARCH/

- executables are copied into `bin/` and renamed in *oldname*`.orig`
- libraries are copied into `lib/`
- for all executables and libraries, TMK determines the necessary shared libraries (via a tool such as `ldd`), and copies them
- TMK creates a script named `dist_wrapper` in `bin/`
- instead of every original executable in `bin/`, a symbolic link is created that points to `dist_wrapper`

Now, if you call for example `bin/myexecutable`, this will follow the symbolic link and call the wrapper script. The script determines the directory it that is in, and then sets the runtime library path to the `lib/` subdirectory. Since the script has been called via a symbolic link, it can determine which executable it is supposed to execute, and so it finally calls `myexecutable.orig`.

This simple mechanism allows to set a number of envinronment variables or different things before calling an executable. Since all executables are called via the same wrapper, it is really easy to maintain and control the distribution.

In addition to what we have explained so far, the `dist` module also provides methods for copying files or file trees, exclusing library trees from being included in the distribution (e.g. directories that only contain system-internal libs), and add code to the wrapper script. Please refer to the reference manual or to the examples to learn more about this.

6.7 Related Modules

Usually software development does not only make use of a single compiler and linker, but also of a number of conversion or code generation tools and external libraries or tools. TMK's module mechanism is a convenient means for providing these additional features to the user in a transparent way. In the following, we shortly sketch some of the C/C++-related modules included in the TMK distribution.

Please note that in general, all these modules should be loaded *after* loading the language module (like `c` or cxx), since sometimes one of the modules must know in advance what linker will be used or for which language it is supposed to generate code.

. . . sorry, not finished yet . . .

6.7.1 The QT Library

6.7.2 The PCCTS Parser Generator

6.7.3 Lex/flex and YACC/bison

Literature

[1] system manual page for `make`. (type ’man make’ on UNIX systems).

[2] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[3] Hartmut Schirmacher. *Installing & Configuring tmk*. `http://www.tmk-site.org/doc/`.

[4] Hartmut Schirmacher and Stefan Brabec. *TMK Reference Manual*. `http://www.tmk-site.org/doc/`.

[5] Hartmut Schirmacher and Stefan Brabec. tmk web pages. `http://www.tmk-site.org`.

[6] Scriptics Corporation. Annotated tcl/tk book list. `http://dev.scriptics.com/resource/doc/books`.

[7] Scriptics Corporation. Tcl/tk tutorial list. `http://dev.scriptics.com/resource/doc/start`.

[8] Brent Welch. *Practical Programming in Tcl and Tk*. Prentice Hall, 1999.

[9] Brent B. Welch and Dave Zeltserman. *The Complete Tcl/Tk Training Course*. Prentice Hall, 1998.

[10] J. A. Zimmer. *Tcl/Tk for Programmers With Solved Excercises That Work With Unix and Windows*. IEEE, 1998.

Index