The tmk Automation Tool

Hartmut Schirmacher, Stefan Brabec

Max-Planck-Institut für Informatik
Im Stadtwald, 66123 Saarbrücken, Germany
http://www.mpi-sb.mpg.de/
{schirmacher,brabec}@mpi-sb.mpg.de

note: this is an early draft version,
not consistent with current tmk application

Contents

Chapter 1

Introduction and Overview

The name `tmk` stands for "TCL-based make". It tries to combine the flexibility and power of the scripting language TCL[1] on the one hand, and the simplicity and utility of `make`[2] with its `Makefiles` on the other. `tmk` was greatly inspired by the `amk` program written by Philipp Slusallek[3] at the University of Erlangen.

On the one hand, the `tmk` program has been designed for managing large code trees without writing exhaustive Makefiles for each directory in the tree. On the other hand, the `tmk` user is able to write very flexible and powerful scripts in order to automate special tasks in certain projects or subprojects.

Since most projects consist of tasks which are repeated over and over and in a similar fashion in many different directories, `tmk` tries to simplify the specification of these tasks by providing

- a convenient specification and scripting **language**
- a simple way of specifying **targets**, **dependencies**, and building **rules**
- a **module concept** which provides a way of specifying common rules and automatically generating targets in an easy and transparent way
- a set of **predefined modules** for handling the most common tasks like compilation of C and C++ code and linking of libraries and executables

Since `tmk` can be extended very easily, and since these extensions can be kept orthogonal to each other by means of the module concept, it is suited for performing any task which has to to with writing scripts and calling compilers, converters, and other programs in order to assemble target files from a number of source files.

[1] see for example John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison Wesley, 1994

[2] see 'man make'

[3] email: `slusallek@informatik.uni-erlangen.de`

tmk requires some basic understanding of the TCL language, since the controlling files are written as TCL code. Section 1.2 gives a brief introduction to the most important concepts of how to specify lists and how TCL will expand expressions containing variables, quotes, and escape characters.

1.1 How to Read This Manual

This documentation is organized as follows:

- Chapter 1 contains the introduction into the basic ideas of tmk as well as the basic rules which define the TCL language, which is the basis of all description files driving tmk's behaviour.
- Chapter 2 uses some simple examples to demonstrate creating a project, compiling, and linking with tmk.
- Chapter 3 provides a complete documentation of tmk's core features (those which are not implemented in a module, but in the tmk program). These features contain the manual specification of targets, rules, and dependencies, as well as subdirectory processing, multiple architecture support, project makefiles, and so on.
- Chapter 4 gives an introduction as well as an in-detail documentation of the tmk module mechanism and the function of the provided modules. It also gives some hint of how to write your own modules. Chapter A gives some details about additional functions provided for operating on lists of files, executing system commands, logging, and handling target names transparently.
- Chapter 5 tells you how to install tmk on your system and how to configure some of the basic mechanisms so that they fit into your personal environment.
- The appendix contains indices of all tmk command line options, global variables, environment variables, and built-in functions.

As a novel tmk user, you should start with the TCL language basics presented in 1.2, followed by the "getting started" in Section 2.

After that, your further reading depends on what kind of job you want tmk to do for you. In most cases, you're going to use tmk for building executables and libraries, so you should try to understand in more detail how to use the default module (cf. Sec. 4.1). But you might as well continue with any other module description from Chapter 4.

If you're planning to dive deeper into the mechanisms of tmk, you can read about the "tmk core" (Chapter 3). However, this is only necessary if you want to track what tmk is doing with your files, or if you want to write your own custom rules and targets.

For those people how want to implement a new module, it would be a good thing to browse through some of the other modules' description in order to get a feeling for the general idea of a module. Then the only really advisable thing is to have a look at the module's source files and create new code by learning from the existing modules. The list of variables and functions in the appendices might prove useful to that end.

1.2 TCL Basics and Expression Evaluation

Since `TMakefile` relies heavily on the features of the TCL language, it is useful to know some basics about TCL. Most of the following page is a reformatted version of the manual page that you can get using `man tcl`. The following rules define the syntax and semantics of the Tcl language:

[1] A Tcl script is a *string* containing one or more *commands*. Semicolons and newlines are command separators unless quoted as described below. Close brackets are command terminators during command substitution (see below) unless quoted.

[2] A command is evaluated in two steps. First, the Tcl interpreter breaks the command into *words* and performs substitutions as described below. These substitutions are performed in the same way for all commands. The first word is used to locate a command *procedure* to carry out the command, then all of the words of the command are passed to the command procedure. The command procedure is free to interpret each of its words in any way it likes, such as an integer, variable name, list, or Tcl script. Different commands interpret their words differently.

[3] Words of a command are separated by *white space* (except for newlines, which are command separators).

[4] If the first character of a word is *double-quote* (") then the word is terminated by the next double-quote character. If semicolons, close brackets, or white space characters (including newlines) appear between the quotes then they are treated as ordinary characters and included in the word. Command substitution, variable substitution, and backslash substitution are performed on the characters between the quotes as described below. The double-quotes are not retained as part of the word.

[5] If the first character of a word is an open *brace* ({) then the word is terminated by the matching close brace (}). Braces nest within the word: for each additional open brace there must be an additional close brace (however, if an open brace or close brace within the word is quoted with a backslash then it is not counted in locating the matching close brace). No substitutions are performed on the characters between the braces except for backslash-newline substitutions described below, nor do semicolons, newlines, close brackets, or white space receive any special interpretation. The word will consist of exactly the characters between the outer braces, not including the braces themselves.

[6] If a word contains an open bracket ("[") then Tcl performs *command substitution*. To do this it invokes the Tcl interpreter recursively to process the characters following the

open bracket as a Tcl script. The script may contain any number of commands and must be terminated by a close bracket ("]"). The result of the script (i.e. the result of its last command) is substituted into the word in place of the brackets and all of the characters between them. There may be any number of command substitutions in a single word. Command substitution is not performed on words enclosed in braces.

[7] If a word contains a dollar-sign ($) then Tcl performs *variable substitution*: the dollar-sign and the following characters are replaced in the word by the value of a variable. Variable substitution may take any of the following forms:

- `$name`: `name` is the name of a scalar variable; the name is terminated by any character that isn't a letter, digit, or underscore.
- `$name(index)`: `name` gives the name of an array variable and `index` gives the name of an element within that array. `name` must contain only letters, digits, and underscores. Command substitutions, variable substitutions, and backslash substitutions are performed on the characters of `index`
- `${name}`: `name` is the name of a scalar variable. It may contain any characters whatsoever except for close braces.

There may be any number of variable substitutions in a single word. Variable substitution is not performed on words enclosed in braces.

[8] If a backslash (\) appears within a word then *backslash substitution* occurs. In all cases but those described below the backslash is dropped and the following character is treated as an ordinary character and included in the word. This allows characters such as double quotes, close brackets, and dollar signs to be included in words without triggering special processing. The following table lists the backslash sequences that are handled specially, along with the value that replaces each sequence.

- `\a`: Audible alert (bell) (0x7).
- `\b`: Backspace (0x8).
- `\f`: Form feed (0xc).
- `\n`: Newline (0xa).
- `\r`: Carriage-return (0xd).
- `\t`: Tab (0x9).
- `\v`: Vertical tab (0xb).

- \<newline>whiteSpace: A single space character replaces the backslash, newline, and all spaces and tabs after the newline. This backslash sequence is unique in that it is replaced in a separate prepass before the command is actually parsed. This means that it will be replaced even when it occurs between braces, and the resulting space will be treated as a word separator if it isn't in braces or quotes.

- \\: Backslash (\).

- \ooo: The digits ooo (one, two, or three of them give an eight-bit octal value for the Unicode character that will be inserted. The upper bits of the Unicode character will be 0.

- \xhh The hexadecimal digits hh give an eight-bit hexadecimal value for the Unicode character that will be inserted. Any number of hexadecimal digits may be present; however, all but the last two are ignored (the result is always a one-byte quantity). The upper bits of the Unicode character will be 0.

- \uhhhh The hexadecimal digits hhhh (one, two, three, or four of them) give a sixteen-bit hexadecimal value for the Unicode-character that will be inserted.

Backslash substitution is not performed on words enclosed in braces, except for backslash-newline as described above.

[9] If a hash character (#) appears at a point where Tcl is expecting the first character of the first word of a command, then the hash character and the characters that follow it, up through the next newline, are treated as a comment and ignored. The comment character only has significance when it appears at the beginning of a command.

[10] Each character is processed exactly once by the Tcl interpreter as part of creating the words of a command. For example, if variable substitution occurs then no further substitutions are performed on the value of the variable; the value is inserted into the word verbatim. If command substitution occurs then the nested command is processed entirely by the recursive call to the Tcl interpreter; no substitutions are performed before making the recursive call and no additional substitutions are performed on the result of the nested script.

[11] Substitutions do not affect the word boundaries of a command. For example, during variable substitution the entire value of the variable becomes part of a single word, even if the variable's value contains spaces.

Chapter 2

Getting started

This section provides a quick start introduction to some of `tmk`'s features by creating a simple example project. The system commands and file naming conventions are for UNIX-style operating systems.

2.1 C++ Compiling in a Single Directory

Suppose you have a directory containing the C++ source files `a.C`, `b.C`, `c.C`, and `myprog.C`, and you want to compile them all into a single executable program, with `myprog.C` providing the `main()` routine. If `tmk` is installed on your system, you simply need to provide a file named `TMakefile` containing the following lines:

```
module cxx
lappend PROGRAMS myprog
```

The first line calls the `module` command which reads the `cxx` module and apply rules and target detection for the C++ language. The second line appends the word "myprog" to the variable `$PROGRAMS`, treating the contents of the variable as a list. This way `tmk` is instructed to build an executable program `myprog` from the object file `myprog.o`.

Having created this file in your directory, you just call `tmk` without any arguments, and the following things will happen:

- A target directory for your current architecture and code level will be created. E.g. if you're working under IRIX 6.5 and with the default code level, this will be the directory `IRIX6.5/`. `tmk` will try to place all generated files in that directory (sometimes this doesn't work because a compiler cannot be convinced to put everything where you want to).
- All `.C` files will be compiled into their corresponding `.o` files using the standard C++ compiler directives.

- All `.o` files except myprog.o will be put into a shared library named `lib`*`directory`*`.so`.
- The executable `myprog` is created by linking together `myprog.o` and the library `lib`*`directory`*`.so`.
- For each `.o` file, a corresponding `.dep` file will be created in the target directory which contains dependency information about the file. This is used to determine when it is necessary to rebuild a target if some source file has changed.

The library containing all object files not corresponding to an executable is called *local* library. The default behaviour of `tmk` is to generate one such local library for every project directory containing one or more suitable object files.

The executable you're building may also depend on some external library, like the math library. In order to link the math library and some other lib `mylib` to your executable, you just specify an additional line

```
lappend SYSLIBS "m mylib"
```

and `tmk` will add the linker directives for linking the libraries to the executable `myprog`. If some library cannot be found in one of the standard library locations of your system, you can additionally specify where to find such libraries:

```
lappend LIBPATH $env($HOME)/mylibs/
```

This will generate the linker options for finding the libraries at link time as well as at run time (for shared libraries). The expression `$env($HOME)` evaluates to the value of the environment variable `$HOME` set by the shell `tmk` was started from.

If you modify one of the source files, e.g. `b.C`, and call `tmk` again, `tmk` will recognize that `b.C` has changed (by means of the file modification time), and only recompile `b.o` and relink the executable. In a similar way, `tmk` recognizes when some of the included header files have changed (through the `.d` dependency files), and recompiles the corresponding object files.

Another feature when using the architecture-dependent target directories as described above is the simple way of cleaning up such a dirctory. If you type

```
tmk clean
```

the directory `IRIX65/` will simply be deleted, removing all automatically generated targets.

2.2 A Simple Project Tree

Now let's assume that you have a project consisting of several directories under some root directory. In this case, you create a file `TMakefile.proj` in the tree's root containing

global variable declarations for the whole project (like compilation flags etc.). The location of the global project file defines the value of the variable `$PROJDIR`, which is used to access the root directory of the current project. For a start, `TMakefile.proj` may be empty.

Additionally, you have to put one `TMakefile` in each directory of you project. These `TMakefile`'s may for example contain a line like this:

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

The `subdir` command takes a list of names and tells `tmk` to recursively build targets in all directories in the list in which it can find another `TMakefile`. The TCL command `[glob *]` expands to the list of all files in the current directory. The `-nocomplain` option prevents the output of error messages if no files are present. `subdir` will filter out all names which do not correspond to a directory, or do not contain a `TMakefile`.

Now, if you call `tmk` in any directory within your project tree, `tmk` will first build all existent subdirectories, and then the directory you have called `tmk` in. By typing `tmk -local`, you can prevent `tmk` from recursing into the subdirectories.

In each subdirectory of your project, you may use different modules and define special targets and rules. One feature automatically provided by `tmk` is the generation of a library for each directory consisting of all object files which do not correspond to an executable. If the example from Section 2 takes place in some directory `mydir/`, `tmk` creates a library named `libmydir.so` (or `.a`, depending on the settings) in the target directory. This library would contain all the object files except `myprog.o`.

With this automatic library feature, it is very easy to reuse code from one project directory in another one, since you can simply link the corresponding library to you executable in the other directory. For example, if `$PROJDIR` points to some project directory named `myproj`, and you are in the subdirectory `myproj/a/b/` and want to use the functionality of the library in directory `myproj/x/y/z/`, you just have to specify

```
set PROJLIBS myproj/x/y/z
```

in the `TMakefile` in the `myproj/a/b` directory. This will automatically deduce the correct library path and library name for you (depending on the `$ARCH` setting etc.). Similarly, is `$PROJDIR` is set, the `c` and `cxx` modules will add the parent directory of `$PROJDIR` as an include path. This way, if your project root directory is named `MYPROJ`, you may include specific header files from within other directories in the project like this in your C++ code:

```
#include <MYPROJ/x/y/z/someheader.hh>
#include <MYPROJ/x/y/z/some_other_header.hh>
```

These `#include` statements will still work if you change the root directory of you project one day. Besides from that, it provides an independent name space for your header files. For more information about the automatic linking, library, and project features, please consult Sections 3.10 and 4.1.4.

It is important to note that it requires some conformance of how to organize the source code in order to get the maximum performance with minimal effort out of `tmk`. The most important rules of thumb are:

- Organize your code into subdirectories. Each subdirectory should contain some subproject in the sense that the files are somehow closely related to each other by function. The directory structure should be planned so that it remains constant except for newly added directories.
- Implement all features which should be reusable in simple source files which do not contain a `main()` routine. For testing or applying the code, write an extra source file containing `main()`. If you do so, all the functionality of the directory can be put into the library, and the test program or application is separated from it.
- Try to split your project into the core parts and the applications using the core parts (in separate directories). Then make sure that the core is always compiled before the applications.
- Try to apply a consistent naming scheme for your directories, files and code objects. This way you may easily replace names by others, your code is more readable, and it is easier to navigate in you code tree.

Chapter 3

The tmk core

The basic process when using tmk is similar to using the well-known make tool. When calling tmk, it searches for a file called TMakefilein the current working directory. In contrast to a standard Makefile, the TMakefile is interpreted sequentially and may contain arbitrary TCL statements in order to define or modify variables, produce output, or do anything you want. In addition to standard TCL commands, several special tmk commands and variables can be used in order to define targets, dependencies, and building rules.

The basic assumption for tmk is that you have a number of *target* files, each *depending* on several *source files* or *dependencies* by means of corresponding *rules*. When the target does not exist, or whenever one or more of the source files change, the target must be rebuilt by executing the appropriate commands. This is the main process for all kinds of development tasks, e.g. the edit-compile-link-test cycle of software development or any similar process involving compilers, translators, filters, etc.

3.1 Specify Which Targets to be Built

The only way to make tmk do anything is to specify which targets you would like to have built. This is basically done by means of the build command:

build <*list of targets*>

This tells tmk to add the specified targets to the list of targets to be processed. This will *not* cause any immediate action. tmk will continue to parse the complete TMakefile before starting to build all the targets. The order of target building is determined by their appearance in the TMakefile, plus the recursive building process which always builds all dependencies before building the dependent target.

Instead of listing the targets in the TMakefile, you can also specify the desired targets at the command line of tmk, overriding the build commands in the TMakefile. If no

target is specified either way or by using the automatic target generation (cf. Sec. 4.1.1), `tmk` will exit with an appropriate message.

If targets are generated automatically, you sometimes need the option of including some files from being build. The `tmk` default module will exclude all targets in the `$EXCLUDE` variable from being built explicitly. If a target appears in a lower level of a dependency chain (meaning it is an indirect target), the `$EXCLUDE` mechanism does not apply.

3.2 Simple Targets and Rules

In order to specify which files a target depends on and how it can be built from those dependencies, the `TMakefile` may contain one or several `target` commands of the form:

`target` <*target*> <*source files*> <*command*>

While we assume <*target*> to be a single file name here, <*source files*> can be an arbitrary TCL list of file names. <*command*> can be any valid TCL script, usually enclosed in braces. A simple example would read like this:

```
target myprog myprog.o {
    cmd CC -o $TARGET $SRC
}
```

Now, if the user types `tmk myprog`, `tmk` will execute the command

```
CC -o <somedir>/myprog <somedir>/myprog.o
```

where `<somedir>` is the directory in which `tmk` puts all automatically generated files by default (cf. Sec. 3.8). The specified command will only be issued if `myprog.o` has changed since the last build or if the target file `myprog` does not exist. You may override this behaviour by means of the `-force` option (cf. Sec. D). The use of the special variables `$TARGET` and `$SRC` is explained in more detail in the next section.

`tmk` command `cmd` is used in order to execute a system user command in a shell (see Sec. A.2). The listed `CC` command will link the object file and produce the executable. `cmd` is similar to the TCL command `exec`, except that it echoes the command and pipes the standard input, output, and error streams to the terminal.

A special case is when <*source files*> is empty (specified as {}). This means that the target has to be built independently of any changes in any source files. So, `tmk` will always try to rebuild it if it comes across this target.

3.3 Multiple Targets and T-Expressions

If multiple targets have to be built by the same kind of commands, it may seem useful to specify a whole *list* of targets and to derive the source file names from the name of the current target. The `target` command in an extended form can be used like this:

`target` <*target files*> <*source files*> <*commands*>

Instead of specifying only a single target, you can use any TCL list of targets. Both <*source files*> and <*commands*> are so-called *target-dependent expressions* or *T-expressions*. As already demonstrated in the previous section, T-expression may contain a number of special variables which are setto target-dependent values before expanding the expression (by means of the TCL `[eval]` command) at building time:

- `$TARGET`: the full target name, including architecture-depending target directories
- `$ROOT`: all characters of `$TARGET` up to (but not including) the last dot (cf. TCL command `[file rootname $TARGET]`)
- `$EXT`: all characters from the last dot on, or the empty string if target name contains no dot (cf. TCL `[file extension $TARGET]`)
- `$DIR`: all characters up to the last slash, or '.' if there are no slashes in the target name (cf. TCL `[file dirname $TARGET]`)
- `$TAIL`: all characters after the last slash, or the empty string (cf. TCL `[file tail $TARGET]`)
- `$BASE`: all characters after the last slash and before the last dot (cf. `[file rootname [file tail $TARGET]]`)

In addition to these variables which are common to all T-expressions, you may use the `$SRC` variable wihtin the <*commands*> argument in order to get a list of all source files. While in this simple example, you could simply use the files explicitly, there are situations where it is imperative to use `$SRC` (e.g. in the case of architecture-dependent targets and a list more than one source files, cf. Sec. 3.8). If you want to pick one of the source files from the list, you may employ TCL commands like `[lindex <list> <index>]` (cf. Sec. 1.2).

It is important to note that both <*source files*> and <*commands*> will be evaluated on the *global* level. This means that all global variables will be known without declaring them explicitly. On the other hand, local variables declared within a procedure or temporarily set within a loop will not be known within the target expression at building time.

A simple example will clarify the use of T-expressions for the target declaration:

```
set CCFLAGS "-g"
target {myprog yourprog} {$ROOT.o someother.o} {
    cmd CC $CCFLAGS -o $ROOT $SRC
}
```

This line in a `TMakefile` tells `tmk` that `myprog` can be built from `myprog.o` and `someother.o`, and `yourprog` from `yourprog.o` and `someother.o`, respectively. For building `yourprog`, `tmk` will execute

```
CC -g -o yourprog yourprog.o someother.o
```

This way you can specify a whole list of files which obey the same rule. Note, however, that `T-expr`'s must be quoted so that the special target-depenedent variables like `$ROOT` will not be expanded when parsing the `TMakefile` (which would result in an error message). For more information about expression evaluation, see Section 1.2.

3.4 Target Patterns

Since there are many standard procedures and naming conventions in the software development cycle, it seems desirable to specify a rule of how to build all instances of a whole *class* of targets. A class of targets may be specified with help of file name patterns as used in the TCL `glob` command and in the file name expansion scheme of most shells. To this end, we have a look at the `target` command in its most general form:

`target` <*target patterns*> <*source files*> <*commands*>

When building the targets, the current target will be compared to each of the patterns specified in <*target patterns*> by means of the TCL built-in `[string match]` command which expands glob-style patterns. Both arguments <*source files*> and <*commands*> work like described in Section 3.3, with the option of using T-expressions.

For example, in order to automatically compile `.C` files into the corresponding `.o` files, the following statement would suffice:

```
target *.o $ROOT.C {cmd CC -o $ROOT.o -c $SRC}
```

Now, whenever a target has to be built which matches the pattern `*.o`, `tmk` will look if there is a corresponding `.C` file and, if needed, will start the compiler in order to produce a new `.o` file.

3.5 Specifying Secondary Dependencies

In addition to the primary dependencies between a target and the source files which are used directly in the building rule, you can provide lots of *secondary* dependencies (e.g. files

included in the source files). For example, if you want to build an object file `x.o` from a C++ file `x.C`, and `x.C` includes some files like `a.h` and `b.h`, you should tell `tmk` that `x.o` should be rebuilt whenever `a.h` or `b.h` have changed, even though the `.h` files do not appear directly in the building rule commands. This can be achieved by means of the `depend` command which has the following syntax:

`depend` <*target*> <*source file list*>

The `depend` command can be understood like a `target` statement without a command argument. Additionally, `depend` only takes a single target argument and does not understand T-expressions. This is due to the fact that `depend` should only be used in the context of target *instances*, not for general rules. Usually, secondary dependencies are generated automatically. In the case of C++ code, the C module makes the compiler dump dependencies into special files and then generates `depend` commands from those files on the fly. See Section 4.2 for a more detailed discussion of this topic.

In order to prevent `tmk` from checking dependencies to system header files and similar code, the user can list file patterns in the variable `$DEPEND_EXCLUDE`. This variable is set to `/usr/include/*` in the `default` module, and will prevent any file below that directory to be checked as a secondary dependency. Just add more paths to be excluded as you like.

It is important to note that in order for a target to be built, all its secondary dependencies must exist (or must have been built). If a secondary dependency does not exist and cannot be built, `tmk` will immediately exit with an error. In contrast, if a *primary* dependency does not exist and cannot be built, this only means that the chosen rule for the current target cannot be applied, and `tmk` will try to apply the next rule. Only if *no* rule can be applied, `tmk` exits with an error.

3.6 Resolving Multiple Rules for One Target

If you specify multiple building rules for the same target, `tmk` will use the first of these rules which is applicable, meaning that all the primary and secondary dependencies for the rule exist or can be built recursively.

When trying to build a target, `tmk` will go through all rules with matching target patterns in the order of their specification. For each rule, `tmk` first tries to find or build all the primary and dependencies. If all primary dependencies are at hand, it checks all the secondary dependencies. If both exist or have been built, `tmk` applies the rule and skips all other rules.

If the primary dependencies (or source files) for some rule cannot be built, `tmk` skips this rule and tries to apply the next one. If no rule is applicable, `tmk` will exit with an error message.

3.7 Dependency Chains

Up to now, we have concentrated on *direct* dependencies. Usually, a building cycle contains whole *chains* of dependencies. For example, some final executable `myprog` will depend on some object file `myprog.obj`, which will again depend on some source file `myprog.C`, and on included files like `myprog.h`. In order to process this chain of dependencies correctly, `tmk` recursively collects all dependency chains for the current target until if finds no more applicable dependency rules. Once the dependencies have been collected, it starts from the bottommost dependency file (the last in the chain) and works its way up to the target. Let's have a look at an example:

```
target myprog myprog.o {exec CC -o $TARGET $SRC}
target myprog.o myprog.C {exec CC $CCFLAGS -c -o $TARGET $SRC}
target myprog.o myprog.c {exec cc $CFLAGS  -c -o $TARGET $SRC}
depend myprog.o myprog.h
build myprog
```

This `TMakefile` explicitly specifies two dependency chains:

1. `myprog` ← `myprog.o` ← {`myprog.C`, `myprog.h`}
2. `myprog` ← `myprog.o` ← {`myprog.c`, `myprog.h`}

`tmk` will start to work its way up from the end of the first chain. It checks if `myprog.h` or `myprog.C` are newer than `myprog.o` (or the `.o` does not exist). If so, it will use the `CC -c` command in order to build the object file. Next, it will compare the date of the object file to that of the executable `myprog`. Again, it will issue the appropriate command if the executable has to be rebuilt. If any of the bottom-most dependencies do not exist (e.g. if there is no `myprog.C`), `tmk` will not be able to build the target using the first chain. So it will try the second chain, requiring a `.c` file. If this file exists, the target will be built from it. If not, and if no other applicable chains exits, `tmk` will exit with an error.

3.8 Multiple Architecture / Codelevel Support

to be updated soon

`tmk` supports the parallel development for multiple architectures by means of a simple mechanism. If the global variable `$USE_ARCH` is set to 1, the variable `$ARCH` will determine the currently active architecture. `tmk` will automatically place all targets in a directory named `$ARCH/`. If you specify a target file `path/targ`, `tmk` will effectively create the target `path/$ARCH/targ`. The `$ARCH/` directory will be created if it doesn't exist. When specifying a dependency file `path/file`, `tmk` will first look for existance of that file, than for the file `path/$ARCH/file`. This means that concerning the specification of target and

dependency names, the multiple-architecture support is completely transparent to the writer of the `TMakefile` and of `tmk` modules. One must only take care that the architecture names do not interfere with directory names reserved for other purposes.

Since all generated targets reside in a single directory, it is very easy to "clean up" a directory. If `$USE_ARCH` is true, the default module (see Sec. 4.1) will define the target `clean`, which will simply perform the UNIX command `rm -rf $ARCH` in the current directory. So the command

```
tmk clean
```

will remove all targets which can be rebuilt by calling `tmk`.

In order to support architecture-dependent source coding, each module should support passing of the `$ARCH` variable to the compilers and linkers in order to allow for architecture-dependent code compilation (see example below).

In order to define rules which depend on the underlying architecture, you simply use conditional TCL expressions in conjunction with the `$ARCH` variable, e.g. :

```
if $USE_ARCH {
  append CCFLAGS " -DARCH_$ARCH"
}

if { "$ARCH" == "IRIX6.5" } {
  set myCC [exec which CC]
} else {
  set myCC [exec which g++]
}

[...]

target *.o {$ROOT.C} {cmd $myCC $CCFLAGS -c -o $TARGET $SRC}
```

In this example, if the current architecture is set to `IRIX6.5`, the compiler will define the macro `ARCH_IRIX6.5` so that C++ code parts may be placed within compile-time conditional statements like this:

```
#ifdef ARCH_IRIX65
    [...]
#endif
```

The currently active architecture can be set at the `tmk` command line using the `-arch` option. The default architecture is determined from the global `tmk` configuration installed on your system (see Section 5).

The $ARCH name is normalized after parsing the tmk command line options, and once again before parsing the TMakefile. Normalization in this context means that all outer spaces as well as all trailing slashes will be removed.

If you want to pass target names to external commands outside of a target command, you need to determine the correct target location by yourself. To this end, you may use the functions shortTargetName and fullTargetName as described in Section A.3.

3.9 Processing Subdirectories

Since most non-trivial projects are organised in tree-like directory structures, tmk directly supports recursive subdirectory processing, including features like environment variable passing and parallel processing of subdirectory lists.

3.9.1 Processing and Excluding Subdirectories

Since it is assumed that subdirectory contain smaller parts of the targets in the current directory, subdirectories are always processed *before* the current directory. In order to specify which subdirectories will be processed, just declare them like this:

```
subdir <subdir list>
```

tmk will first exclude all directories from the <*subdir list*> which are contained in the $SUBDIR_EXCLUDE variable. Then, it will filter out all names which do not correspond to an existing directory by means of the TCL [file isdirectory] command. Next, it will skip all directories which do not contain a TMakefile. This is especially important if you build targets for multiple architectures (cf. Sec. 3.8), since tmk must distinguish genuine code directories from the self-generated $ARCH directories.

Only the remaining directories will be processed. Tilde expressions will be correctly expanded (see TCL file command man page). Subdirectory processing will take place at exactly the time when the subdir command is being processed during parsing the TMakefile.

For example, if you always want *all* current subdirectories to be processed, you may include the following line:

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

The argument expression determines the list of all files in the current directory (possibly none, without generating an error). tmk will automatically filter out everything which is not a directory or does not contain a TMakefile, and so it will descend only in the reelevant directories.

When processing the specified subdirectories, the program $TMK will be called with the command line arguments in $ARGS. $ARGS will be filled with the arguments passed to the current instance of tmk.

TCL variables are *not* passed down to the `tmk` subprocesses. This is mainly due to the fact that an invocation of `tmk` in a subdirectory should normally do the same thing as if `tmk` would have been invoked recursively from within a parent directory.

However, if you want to pass variables down to the subprocess, you can employ the environment variable mechanism by means of the TCL `env` array. Use the variable name `$env(SOMEVAR)` in order to access the environment variable `$SOMEVAR` in the current environment.

3.9.2 Directory-Based Parallel Processing

Since often subdirectories can be processed independently of each other, `tmk` provides a way of specifying that certain directories can be processed in parallel. The user simply specifies a list of *lists of directories* instead of a simple list:

`subdir` <*list-of-lists of subdirectories*>

All directories of each *inner* list will be processed in parallel. A simple example would read like this:

```
subdir {{testA testB} {testC testD testE}}
```

This would cause the parallel building of directories `testA` and `testB`, and then the parallel processing of the other three specified directories. *Parallelism not implemented yet.*

3.10 Project Makefile

Unless called with the `-noproj` option, tmk will go from the current directory upwards until it reaches `/` or finds a file named `TMakefile.project`. This file will be processed before the local `TMakefile` in order to allow project-wide "global" definitions and functions. During parsing of the project makefile, `tmk` will set the current working directory to that of the project file. So it is very easy to store the project root directory in a variable like this:

```
set PROJDIR [pwd]
```

If you want to use an alternative file, you may explicitly specify a project makefile by means of the command line option `-proj <filename>`.

Set setting of `$PROJDIR` also enables the use of a lot of mechanisms like comfortable specification of include paths and project libraries, as defined in the default module. Please refer to the next section for details on this topic.

3.11 Debugging

When you encounter an error or a strange behaviour that you (or your systems administrator) cannot explain, you may want to try the `-debug` option of `tmk`. This causes `tmk` to output lots of verbose comments about what it is acutally doing and why. The best way to debug a `tmk` session is to pipe the debugging output into a file and then search in this file using a text editor program. For example, if you're using the EMACS system, you can start `tmk -debug` using the compile command and then look at the output in the resulting EMACS buffer.

First, `tmk` gives some information about the configuration it uses, e.g.:

```
tmk:  [dbg] machine:  mips sgi IRIX 6.5 UNIX
```

Then it tells which files are being read and processed, what the project directory is set to, and so on:

```
tmk: [dbg] found /usr/htschirm/proj/IBR/TMakefile.proj
tmk: [dbg] setting PROJDIR to /usr/htschirm/proj/IBR
tmk: [dbg] found /usr/htschirm/proj/IBR/TMakefile.priv
tmk: [dbg] reading module /usr/htschirm/proj/tmk/modules/default.tm
tmk: [dbg] ----- begin processing /usr/htschirm/proj/IBR/TMakefile.
----
tmk: [dbg] reading module /usr/htschirm/proj/tmk/modules/newclass.t
tmk: [dbg] ----- end processing /usr/htschirm/proj/IBR/TMakefile.pr
----
tmk: in directory /usr/htschirm/proj/IBR/filter/single
tmk: [dbg] ----- begin processing TMakefile -----
tmk: [dbg] reading module /usr/htschirm/proj/tmk/modules/cxx.tmk
tmk: [dbg] adding target/rule *.o <- {$ROOT.C}
tmk: [dbg] reading dependency files..
[...]
```

Furthermore, `tmk` protocols every target or dependency that is being added, and it also shows the dependencies that are excluded via `$DEPEND_EXCLUDE`. So if you're tracking some special file for which the building process does not seem to work, you can simply search for the filename in the output protocol, and you will see in which dependency chains and rules it is involved. The example below shows the effect of two `target` commands and a `build` command.

```
tmk: [dbg] adding target/rule *.o <- {$ROOT.c++}
tmk: [dbg] adding target/rule libIBR_filter_single.so <-
           toFloat.o Quantize.o Crop.o [...]
tmk: [dbg] adding default targets: libIBR_filter_single.so
```

After having built the rule database for the current directory, `tmk` proceeds with working from the specified toplevel targets down to the lowest dependencies. For each target, it lists which primary and secondary dependencies there are, and then it recursively checks all those dependencies.

```
tmk: [dbg] toplevel targets: libIBR_filter_single.so
tmk: [dbg] checking target: libIBR_filter_single.so
tmk: [dbg] prim dep for libIBR_filter_single.so: toFloat.o Crop.o [..
tmk: [dbg] sec  dep for libIBR_filter_single.so:
tmk: [dbg] checking target: toFloat.o
tmk: [dbg] prim dep for toFloat.o: toFloat.C
tmk: [dbg] sec  dep for toFloat.o: [...]
tmk: [dbg] toFloat.C: no matching rules/dependencies, but exists.
tmk: [dbg] Crop.o must be built because it does not exist.
tmk: [dbg] IRIX6.5/libIBR_filter_single.so must be built because
           IRIX6.5/Crop.o has been updated
```

In the above exmaple, `tmk` checks the dependencies for the specified library. The source file `toFloat.C` exists, and since no secondary dependency for `toFloat.o` is newer than the object file, the object file will not be rebuilt. In contrast, `crop.o` does not exist yet, and so `tmk` is going to build it from `Crop.C`. As a result of this, the library containing `Crop.o` must also be rebuilt.

When checking the dependencies for the current target, tmk looks up the dependent targets in its internal cache. If it has already checked the status of that target, it will output the "cache hit" (including the coded file modification time). If not, it will say something about the target's status and add it to the cache.

If `tmk` returns from processing the dependencies for a target, it outputs something like

```
tmk:  [dbg] back to processing RemoveBiasScale.o
```

This way you should be able to find your way through `tmk`'s debugging output.

Chapter 4

Modules

Modules provide a way to predefine a huge number of general or specific rules and using them effectively. A module is nothing more than a file containing statements in the same syntax as in a `TMakefile`. This means it can define variables and procedures, declare rules and dependencies, add targets to the list of to-be-built targets, produce output, and so on. You can invoke a module by means of the `module` command:

`module` <*list of modules*>

The <*list of modules*> may contain any number of names. For each module <*name*> `tmk` will look for the corresponding file in the following places:

1. `./`<*name*>`.tmk`
2. for each element <*path*> in the list `$env(TMK_MODULE_PATH)`: <*path*>`/`<*name*>`.tmk`
3. <*directory where* `tmk` *is installed*>`/modules/`<*name*>`.tmk`

The second way of specifying a module path can either be used by setting the environment variable `TMK_MODULE_PATH` from the shell, or by putting a statement like

```
set env(TMK_MODULE_PATH) somepath/
```

into your `TMakefile`.

For enhancing the power of `tmk`, it comes with a number of predefined modules which will be described in the following sections. They all share some common features in order to support things like automatic target generation, multiple-architecture support, etc. Some modules do their work if you just call them, others allow to configure them via global variables before or after calling them.

4.1 The `default` Module

By default, `tmk` activates the module `default` which can be found in the `tmk` installation directory under `modules/default.tmk`. This script sets up some global variables and basic routines in order to provide features like target generation for static and shared libraries.

The first thing the default module does is to set some useful variables concerning the current working directory and the project directory:

- `$PROJDIR`: this is set by the `tmk` core; it is the absolute path of where the `TMakefile.proj` has been found, or simply the current working directory if no project file could be found.
- `$PROJROOT`: the parent directory of `$PROJDIR`
- `$SUBDIR`: the path of the current subdirectory, relative to `$PROJROOT`
- `$DIRTAIL`: the last component of the current working directory

So for example, if you're in a directory

`/home/myname/proj/myproj/a/b/c`,

and the project makefile was found in `myproj`, then the following values will be set:

```
set PROJDIR  /home/myname/proj/myproj
set PROJROOT /home/myname/proj
set SUBDIR   myproj/a/b/c
set DIRTAIL  c
```

4.1.1 Automatic Target Generation and Exclusion

Automatic target generation means that some `tmk` modules like `C` and `c` look for source code files in your directory and automatically append the corresponding `.o` files to the `$AUTOTARGETS` variable. For example, the `cxx` module will look for all files with any of the suffixes listed in `$CCEXTENSIONS` and will register the corresponding `.o` files as automatic targets. You can switch this mechanism on and off by setting the flag `$MAKE_AUTOTARGETS`. The default is on. All modules which support this feature will append their module-specific targets to the `$AUTOTARGETS` variable. Lateron, the `$AUTOTARGETS` list is used for more mechanisms like library generation, automatic linking, and so on (cf. next section).

If `tmk` is called with an explicit target argument, the automatic target detection proceeds as usual, but only the explicitly listed targets will be built.

In order to exclude some files from being processed as an automatic target, you may set the `$EXCLUDE` variable to any target you do not want to have included in the automatic

target list. Please note that in the case of compiler targets, you must specify the object file (e.g. `.o`) as an autotarget, not the source file (like `.C`).

After the `TMakefile` is completely processed, a default module procedure (set up using `beforeBuilding`, cf. Sec. A.2) will automatically remove the `$EXCLUDE` targets from the `$AUTOTARGETS` list.

4.1.2 Library Specification

The `tmk` default module provides some mechanisms for specifying libraries which should be linked with the executables (or even with the generated libraries) in the current directory. For "external" or "system" libraries, you must provide the symbolic library names and (if needed) a list of paths in which to search for those files:

```
lappend LIBPATH $env(HOME)/mylibs
lappend SYSLIBS m xt mylib1 mylib2
```

In this example (and for IRIX systems), `tmk` will first search for the files `libm.so`, `libm.a`, `libxt.so`, `libxt.a`, `libmylib1.a`, ..., in the paths specified in `$LIBPATH`. Each file that will not be found in any of the `$LIBPATH` directories will be searched for by the linker in its default locations.

The problem with this way of library specification is that there may not be two libraries with the same name, since the linker would always find the first one in the path and then try to resolve all symbols using that library. So when using `tmk`, "project", or "internal", libraries are automatically assigned names which are unique within the complete project tree. For example, if you have created a library in some directory `a/b/c/` (relative to the root of the project tree), you can specify that library uniquely by the command

```
lappend PROJLIBS a/b/c
```

This tells `tmk` to look in the directory `a/b/c` for a library called `a_b_c`. This way most ambiguities are avoided easily. For example, if you're using the architecture-dependent compilation (targets are put into an `$ARCH` directory) and if you're on an IRIX system, the actual library file name `tmk` will look for is

```
$PROJROOT/a/b/c/$ARCH/liba_b_c.so
```

for a shared library, or

```
$PROJROOT/a/b/c/$ARCH/liba_b_c.a
```

for a static library.

In order to make the specification of libraries complete, there are two more parameters. The variable `$LINK_MODE` specifies one of four possible linking modes. The choices are "static_only", "shared_only", "static_first", and "shared_first", depending on whether you

want to enforce the use of static or shared libs, or whether you want to give a preference (e.g. if a shared version of a lib exists, then link that one; else link the static version). If the flag `$LINK_LIB_TWICE` is on, then all libraries will be specified twice for each link command. This helps resolving most of the problems with inter-library dependencies.

So a complete specification of libraries for a linking command needs five parameters:

- system lib path
- system lib names
- project libs
- link mode
- link-twice flag

The function LIBSPEC takes these five parameters and compiles them into a *library specification* list, as it is done with `$LIBPATH`, `$SYSLIBS`, `$PROJLIBS`, `$LINK_MODE`, and `$LINK_LIB_TWICE` for the default targets.

4.1.3 Multiple Project Locations

The default module also provides a way of linking project libraries from different project locations. That means that you can have parts of the project in your home directory, and other parts reside in some central directory which is shared by all users of those project parts. `tmk` will automatically link libraries from the central version whenever it does not find the corresponding project subdirectory in you primary project tree.

The location of the primary project tree is defined by the `$PROJROOT` variable, which is set according to the parent of `$PROJDIR`, the directory where you project makefile has been found. In addition, you may set the variable `$PROJ_LOCATIONS` to an arbitrary number of paths pointing to different project root directories.

For generating dependencies and for the linking of every single project library(`$PROJLIBS`), the default module will look for the corresponding project subdirectories (not including `$ARCH`) in the `$PROJROOT` hierarchy first. If the directory is found, then `tmk` will assume that this project lib has to be there lateron. If not, then `tmk` looks in the directory hierarchies specified by `$PROJ_LOCATIONS`, proceeding in the specified order. The first project subdirectory that actually exists will be used for the dependencies and linking.

Note that if you change the location of some part of your code tree, you usually will have to rebuild all dependency files, since these contain absolute paths due to performance issues. You can do this by simply building the `depend` pseudo-target.

4.1.4 Local Library Generation

After the `TMakefile` has been parsed, the `default` module removes all those targets from `$AUTOTARGETS` which are specified via `$EXCLUDE`. Then, it extracts all object files from `$AUTOTARGETS` and stores them in the `$AUTO_OBJ` list. Next, `$PROG_OBJ` is created, containing the object files corresponding to the targets specified in `$PROGRAMS`. Finally, the object files to be put into the library (`$LIB_OBJ`) are determined by choosing those files from `$AUTO_OBJ` which are not contained in `$PROG_OBJ`.

Depending on the flags `$MAKE_STATIC_LIB` and `$MAKE_SHARED_LIB`, `tmk` creates a static and/or static library in the current directory, containing the `$LIB_OBJ` object files. If the flag `$LINK_LIB_INTO_LIB` is on, then the libraries will be linked with additional libraries, as specified by the variables `$LIBPATH`, `$SYSLIBS`, `$PROJLIBS`, `$LINK_MODE`, and `$LINK_LIB_TWICE` (cf. Section 4.1.2).

The name of the libraries is determined by the path from the project's parent directory down to the current subdirectory. For example, if you're in the directory `myproj/a/b`, the libraries will get the symbolic name `myproj_a_b`. The actual filename depends on the operating system. For IRIX, the static lib would be called `libmyproj_a_b.a`, and the shared lib `libmyproj_a_b.so`.

The result of this mechanism is that as default, there will be one (shared) library in each subdirectory of your project, each containing the functionality of that directory. This can be used very conveniently to put together your code piece by piece, since the libraries can be identified uniquely by position in the code tree

4.1.5 Executable Generation

As mentioned before, the variable `$PROGRAMS` will specify which executables shall be built. For every executable (e.g. `x`), one target will be generated which links the object file (`x.o`) with the libraries specified by `$SYSLIBS`, `$PROJLIBS`, etc. (cf. Sec. 4.1.2), plus the local library containing all the other object files from the current subdirectory (cf. Sec. 4.1.4). If no local library is built (e.g. both `$MAKE_STATIC_LIB` and `$MAKE_SHARED_LIB` are 0), then the executable will be linked with the object files directly[1].

4.1.6 Default Module: Example

Let's assume the following settings:

```
set SUBDIR            a/b/c
set MAKE_SHARED_LIB   1
set MAKE_STATIC_LIB   0
set LIBPATH           {path1 path2}
```

[1] Actually, this is currently not implemented, but will be provided on request. Sorry.

```
set SYSLIBS           {m xt}
set PROJLIBS          {a/x/y/c a/x/z}
set LINK_MODE         "shared_first"
set LINK_LIB_TWICE    1
set LINK_LIB_INTO_LIB 1
set PROGRAMS          {exec1 exec2}
set EXCLUDE           {c.o}
module C
```

Now let's assume further that there are the source files `exec1.C`, `exec2.C`, `a.C`, `b.C`, and c.C. Now the `C` module will set

```
set AUTOTARGETS {exec1.o exec2.o a.o b.o c.o}
```

with rules for compiling the `.C` files into `.o` files. The `default` module now sets:

```
set AUTOTARGETS {exec1.o exec2.o a.o b.o}
set AUTO_OBJ {exec1.o exec2.o a.o b.o}
set PROJ_OBJ {exec1.o exec2.o}
set LIB_OBJ  {a.o b.o}
set libspec  [LibSpec {path1 path2} {m xt} {a/x/y/c a/x/z}\
              "shared_first" 1]
```

With these definitions at hand, it will generate the following top-level targets:

- make shared lib `liba_b_c.so` containing `$LIB_OBJ`, and link it with the libraries specified by `$libspec`.
- make executables `exec1` and `exec2` from the files `exec1.o` and `exec2.o`, repsectively. Link them with `liba_b_c.so` and the libs given by `$libspec`.

4.1.7 Enforce Building of a Target

Sometimes, you may want to enforce the building of a certain target. One way of doing this is to create a target without dependencies, e.g.

```
target x {} {...}
```

Anyway, sometimes the target in question will need some argument `y` passed as dependency. So in this case the above example cannot be applied. For this purpose the default module defines the pseudo target `force_rebuild` which can be used to enforce the creation of target `x` like this:

```
target x {y} {...}
depend x force_building
```

4.1.8 Cleaning Up

The default module defines the default target `clean`, which is used for cleaning up all automatically generated and temporary files in a directory. If you're not using the multiple architecture support (`$USE_ARCH` is 0), `tmk` executes the command

```
rm -rf [glob -nocomplain $CLEAN_PATTERNS]
```

If `$USE_ARCH` is 1, then the command reads

```
rm -rf [glob -nocomplain $ARCH/ $CLEAN_PATTERNS]
```

If you also want other files to be deleted when calling `tmk clean`, simply append them to the `$CLEAN_PATTERNS` variable, e.g.

```
lappend CLEAN_PATTERNS *~
```

4.1.9 Regenerating File Dependencies

Another important target is defined for updating all dependency files for all targets. This can be necessary when due to a version update depencies change from one file to another one (e.g. by renaming an include file). The `depend` target is a pseudo-target which basically does nothing. For each object file `$xyz.o` to be generated, the language modules (like `c` and `cxx`) will append dependency-generation targets `$xyz.depend` to the `$AUTODEPEND` list. The default module then excludes dependency files correspdonding to targets in the `$EXCLUDE` list, and for each remaining dependency target, makes this target a prerequisite of the global `depend` target, e.g.:

```
depend depend xyz.depend
depend xyz.depend force_building
```

The second line (cf. Sec. 4.1.7 has to be specified to make sure that the dependency will be generated if `depend` is built, regardless of the file's status or age.

The language modules have to define the `xyz.depend` target's functionality, because it depends on the compiler and other language-dependend things how to update the dependency files. In the end, you can update all dependency files by simply calling

```
tmk depend
```

in the corresponding directory tree.

4.1.10 `default` Module Internals

This section presents some basic functions and variables provided by the `default` module. You can use these to easily define targets (executables, libs, shared objects) which do not fit in the default pipeline, but use the same basic routines.

Sorry, the functions are not documented here yet. Please have a look into the file `default.tmk` for more details.

- READDEPENDENCIES
- UNIQUELIBNAME
- LIBSPEC
- MAKETARGET_*
- FILENAME_*
- SEARCHINPATHLIST
- FINDLIB
- FINDPROJLIBDIRECTORIES

4.2 `c` and `cxx`: C/C++ Compilation

The `c` and `cxx` modules provide methods for compiling C and C++ code and automatically generating all corresponding targets and dependencies in the current directory.

4.2.1 Compilation and Dependencies

First, the modules define general rules for compiling a C/C++ source file into an object file. For every source file extension `$EXT` from the list `$CEXTENSIONS` / `$CCEXTENSIONS`, the following rule is crested:

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

For C++, simply replace nearly all `C`'s by `CC`'s. The source file `$SRC` in this case will be the C/C++ file, and `$TARGET` will expand to the object file to be created in the `$ARCH` directory. The `$CC_MDEPENDENCIES` variable contains an expression which will be evaluated in order to generate the compiler's command line options for writing the dependencies into the right dependency file.

Next, rules are generated for renewing a target's dependency file:

```
target *.depend $ROOT.$EXT {
  $C $CFLAGS $C_MDEPENDENCIES $C_DEP_ONLY -c $SRC -o $ROOT.o
}
```

For any target `x`, the corresponding target `x.depend` will be in charge to re-generate the dependency file. The `$C_DEP_ONLY` variable is a compiler switch for not actually compiling the code, but only generating the dependencies.

Now, for manually generating a target for a source file `x.C`, only one function call is needed:

```
MakeTarget_C x.C
```

This will cause the following actions:

- if `$MAKE_AUTOTARGETS` is on, `x.o` will be appended to the `$AUTOTARGETS` list (and thus, `x.o` will be part of the automatic library/executable linking process)
- `x.depend` will be appended to the `$AUTODEPEND` list (cf. Sec. 4.1.9)
- if a dependency file already exists, it is read using READDEPENDENCIES in order to generate the right dependencies within `tmk` (cf. Sec. 4.1.10)

The MAKETARGET_C function returns the target name for generating the object file. So if you want to build an object file manually, just do something like:

```
build [MakeTarget_C x.C]
```

Depending on the project parent directory `$PROJROOT` defined by the location of the project makefile (cf. Sec. 3.10), the `c` and `cxx` modules will automatically add the include path option `-I$PROJROOT` to the compiler flags. If `$PROJ_LOCATIONS` is specified, one additional `--I` statement will be generated for each specified path. This way, include files may be specified relative to the project root directory, regardless of where the corresponding subdirectory is actually located. The proper include statement for an include file `d.h` in the subdirectory `a/b/c` in project `myproj` looks like this:

```
#include <myproj/a/b/c/d.h>
```

The compiler will always choose the file in the first project subdirectory among the `$PROJROOT`/`$PROJ_LOCATIONS` paths that can be found. This way, the include paths do not need to be changed when the project home moves, and on the other hand the include files as well as the libraries are identified uniquely by the path relative to the project root.

4.2.2 Automatic Target Generation

In addition to just defining the rules for C/C++ targets, the `c` and `cxx` modules automatically detect the source files in the current directory, and call the MAKETARGET_C function for those detected source files.

The variable `$CEXTENSIONS` (`$CCEXTENSIONS` for C++) contains all extensions which are assumed to be used for C (C++) code files. At the moment of calling `module c` or `module cxx`, `tmk` will look for files with these extensions and automatically call the MAKETARGET_C fucntion for generating object file and dependency targets and updating the `$AUTOTARGETS` list, if `$MAKE_AUTOTARGETS` is 1[2]. As usual, target files (`.o`) listed in `$EXCLUDE` will not be built.

C++ and Templates

An important issue for compiling C++ code is how to design your code and Makefiles in order to employ meta-code mechanisms as for example templates. Templates introduce a whole number of new constraints and problems to the traditional C++ compilers and linkers, and so this topic is worth an extra amount of consideration.

For the MIPSpro C++ compilers, it is important to obey the following rules in order to make templates work together with libraries and complicated code trees:

- Use a consistent naming scheme. Most importantly, if a template is declared in a file `file.h`, the template implementation (if there is any non-inline code) will be searched by the compiler in the files `file.C`, `file.cpp`, and under similar names. Please consult your compiler manual pages for details.

- Since templates are normally instantiated when they are needed, it may happen that some program using a library may require some library code to be "recompiled" by the prelinker in order to generate template instances for the user program. In order to do this, the prelinker must find the *source files* of that library. If all include files are specified using the project-relative paths as discussed above, the prelinker will be able to find all include files and the corresponding source files.

- You should always use the compiler options `-ptnone` and `-prelink` when compiling C++ code. This ensures that templates will not be instantiated too often for the same program.

- If the flag `$CC_PERLINKER` is on, the `C` module automatically patches the MIPSpro compiler's `ii_files` in such way that the *prelinker* will always run the compiler with

[2]Please note that in order to change the file extensions used for generating the targets, you must modify `$CEXTENSIONS` / `$CCEXTENSIONS` *before* calling the `module` statement. The best way to change the default extensions is to set `$CEXTENSIONS` and/or `$CCEXTENSIONS` in the project makefile, `TMakefile.proj`.

the -DCCPRELINK option. When compiling in "normal" mode, -UCCPRELINK will be used. This provides a simple means of writing code that is compiled only during the prelinking phase or only during the "real" compilation phase, for example by using

```
#ifndef CCPRELINK
   ... code which will NOT appear during prelinking ...
#endif
```

4.3 qt: QT Library / Precompiler

The qt module supports the automatic precompilation of QT-specific header files into so-called "meta object" C++ code. Also, it appends the QT-specific library path to the $LIBPATH variable, and include paths to $CCFLAGS.

Depending on the value of $QTDIR, the qt module will append the --I$QTDIR/include to the $CCFLAGS, and $QTDIR/lib to the $LIBPATH variable. $QTDIR, if not manually specified by the user, is set depending on the flavours N32 and N64.

For precompiling the QT header files, the $QTPATTERNS variable contains glob-style patterns determining which files are supposed to be QT-specific header files. Currently, $QTPATTERNS is set to

```
*.qt.hh *.qt.H qt*.hh qt*.H qt*.h++ *.qt.h++ qt*.h *.qt.h
```

All header files matching one of these patterns will be precompiled by the meta object compiler moc using the command

```
cmd $QTMOC $MOCFLAGS -o $TARGET $SRC
```

where $SRC is the header file, and $TARGET is the target C++ file name, which is the rootname of the header file, plus .moc.C. The corresponding .moc.o file is added to the automatic targets list $AUTOTARGETS. $QTMOC, if not set otherwise by the user, is set to $QTDIR/bin/moc. $MOCFLAGS is initialized with -p [pwd], which is a necessary workaround when generating targets in a different directory, like in $ARCH/.

4.4 yacc: Parser Generator

YACC stands for "yet another compiler-compiler" and is a tool for generating a parser (in C code) from a grammar. The yacc module provides rules for generating C/C++ code from grammars, detects the grammars and registers the corresponding automatic targets.

Before calling the yacc module, you may choose the file suffix which indicates a grammar file to tmk by setting $YACCSUFFIX. Similarly, you can modify the suffix of the code

and header files to be generated by setting the variables $YACCSUFFIX_C and $YACCSUFFIX_H. The defaults are y, C, and hh.

The yacc module generates rules for generating a header and a C code file from each grammar. To this end, it calls the command $YACC with options $YACCFLAGS. Since the resulting header file will have the wrong name, tmk will rename the header file as desired. Both the .C and the .hh file will be placed in the $ARCH directory (if $USE_ARCH is on). For each translated grammar, the corresponding object file (.o) will be added to the list of automatic targets (cf. Sec. 4.1). etc.).

The default $YACC is /bin/bison -y. The default $YACCFLAGS are -d.

4.5 lex: Lexicographical Analyzer

Lex is a lexicographical analyzer, usually used in conjunction with a parser generator like YACC (cf. Sec. 4.4). The lex module generates rules for translating a lex description file into a C code file. The suffix of the lexicographical file can be set via $LEXSUFFIX before calling the module. The suffix of the code file to be generated can be set via $LEXSUFFIX_C. The command to be called can be modified (at any time in the TMakefile) via $LEX and $LEXFLAGS. The defaults are

- $LEX = flex,
- $LEXFLAGS =
- $LEXSUFFIX = l,
- $LEXSUFFIX_C = C

Since the lex file usually needs the include file generated by the parser generator like YACC (cf. Sec. 4.4), the module adds $ARCH to the the include path for the C and C++ compilers (if $USE_ARCH is on).

4.6 doxygen: C/C++ Documentation Generation

Doxygen is a system for automatically generating HTML, LaTeX, and manpage documentation from C++ files[3]. Doxygen uses a configuration file containing several variables which define where to look for the source files, which files to consider, how to name the project, and so on. Unfortunately, this configuration file is static, meaning that there is no way of using environment variables or similar mechanisms inside the Doxygen configuration file. In order to bypass this inconvenience, tmk will modify the config file on the fly before calling doxygen.

[3] see http://www.stack.nl/~dimitri/doxygen

In order to create a documentation for your project `myproj`, you simply add a subdirectory (e.g. `myproj/doc`) and write a `TMakefile` like this:

```
module doxygen
set DXX_PROJECT_NAME     "Name-of-my-Project"
set DXX_INPUT            "$PROJDIR"
set DXX_FILE_PATTERNS    "*.h *.hh *.H *.h++ *.hpp *.hxx *.doxy"
```

The first line instructs `tmk` to load the Doxygen module. The module will do the following things prior to actually generating the documentation:

- generate a Doxygen config file named `$ARCH/$DOXYFILE_INPUT`, using the `$DOXYGEN -g` command
- for each `tmk` variable `$DXX_SOMEVAR`, it will replace the corresponding variable definition `SOMEVAR = <something>` in the config file by the definition provided in the `TMakefile`.
- write the "patched" config file to `$ARCH/$DOXYFILE`

This simple mechanism allows to use the `tmk` variables directly for configuring Doxygen. Since the `TMakefile` is used as the "true" config file, the steps described above will be performed whenever the `TMakefile` is newer than the Doxygen config file.

The module generates the following targets:

- doc: generate HTML, LaTeX, and manpages. This is the default target when calling `tmk` without arguments. If you do not want to generate all versions of the documentation, you may set some of the variables `$DXX_GENERATE_HTML`, `$DXX_GENERATE_LATEX`, and `$DXX_GENERATE_MAN` to `"NO"`.
- ps: first build target `doc`, and then call `make refman.ps` in the LaTeX subdirectory in order to generate a Postscript version of the reference manual.

The documentation will be generated in the directories `html, latex, man` in the `$ARCH` subdirectory. This is quite convenient because Doxygen contains a C++ preprocessor which can generate macro definition-dependent documentation.

4.7 `newclass`: Generate Files From Templates

`newclass` simplifies the creation of header files for new class definitions. Usually, only a single class definition is placed in a header file. So it makes sense to use a template for header files which contains some project-specific header as well as the basic class definition code. If you put the line

```
module newclass
```

in your `TMakefile.proj` (cf. Sec. 3.10), you may call

```
tmk newclass
```

in any of your project subdirectories. `tmk` will ask you for three things:

- the name of the class to be defined
- the template argument list (optional)
- a brief description of the classes' purpose

Then, `tmk` looks for all template files it can find, expands certain expressions in those template files, and creates new code files in your directory. `tmk` looks for all template files `$NEWCLASS_TEMPLATE.*`. `$NEWCLASS_TEMPLATE` defaults to `$PROJDIR/newclass`. The resulting files are named after the class, plus the suffix of the template file.

The template file may contain any number of (non-nested) expressions of the form

```
[@@ expression @@]
```

The `expression` will be expanded by a TCL `[eval]` command, and the whole expression will be replaced by the expanded expression. If your expression contains spaces, you should place it within double quotes.

In addition to this general mechanism, the following variables will be set in addition to all other `tmk` global variables:

- `$CLASSNAME`: name of the class
- `$DESCRIPTION`: brief description of class
- `$FILENAME`: full name of the file to be written
- `$SUFFIX`: suffix of the file to be written
- `$TEMPLATE_ARGS`: template argument list, e.g. `int N, typename T`, possibly empty
- `$TEMPLATE_TYPE`: template argument list enclosed in <>, e.g. `<int N, typename T>`; empty string if `$TEMPLATE_ARGS` is empty
- `$TEMPLATE_DEF`: template definition code, e.g. `template<int N, typename T>`, possibly empty.
- `$USERNAME`: the result of the `whoami` command

- `$CREATOR`: the user's full name (inferred from the `passwd` entry)
- `$CVSID`: simply expands to `$Id$`
- `$CVSLOG`: simply expands to `$Log$`

If the variables `$CLASSNAME`, `$TEMPLATE_ARGS`, and `$DESCRIPTION` are set manually, the user will not be asked for those values. This is useful for "misusing" the module for other, similar tasks.

4.8 `dist`: Make Executable Distributions

With the `dist` module, you can generate a distribution of everything that is needed for "giving away" a standalone executable. This is non-trivial as soon as you're using shared libraries, or dynamic shared objects which are linked dynamically at runtime. You also may want to include example files or non-standard libraries in your distribution, and the resulting software should be able to run in any directory that it is put into. To this end, the `dist` module does the following:

- examine the executables you want to distribute, and find all shared libs needed to run them
- copy the executables and all used libs from within the project tree(s) and from other specified directory trees
- create a wrapper for each executable, so that the executable will run and find its libraries in any environment

It is possible to use some subdirectory in an existing project or create a 'dist' project in your project tree. In the latter case, you will need to define a (possibly empty) `TMakefile.proj`. Then, in order to create a certain distribution, the most important lines of code in you `tmk`are

```
module dist
lappend DIST_TARGETS a/b/c/$ARCH/myexec
```

This tells `tmk` to copy the executable `myexec` for the current architecture from the project directory `a/b/c`. If you specify an absolute path, `tmk` will take the specified executable directly. If you specify a relative path, `tmk` assumes that you specify some executable in a project tree, and will look for the corresponding directory in `$PROJROOT` as well as in `$PROJ_LOCATIONS` (cf. Sec. 4.1.3).

With the selected executable, `tmk` will use the `ldd` system command to determine which shared libraries are needed by the executable. Then, `tmk` determines which of these libraries are either

- in the directory tree below $PROJROOT (cf. Sec. 4.1),
- in a directory tree below any of $PROJ_LOCATIONS (cf. Sec. 4.1.3),
- or below any directory specified in $DIST_COPY_LIB_DIRS

This last variable is empty by default and can be set in the distribution TMakefile. The $PROJ_LOCATIONS are usually set in the TMakefile.proj.

Those libraries which satisfy any of the above conditions will be considered as "copy-worthy" for the distribution, and the dist module will create targets to copy the files if they are newer than the current ones in the distribution.

The directory structure in the distribution target directory looks like this:

- $ARCH/$DIST_BIN/: executables and the wrapper script
- $ARCH/$DIST_LIB/: libraries and shared objects
- default names: bin and lib

This means that you can modify the structure by setting the corresponding variables.

Apart from the executable(s) and the necessary libraries, you may want to copy additional files into the distribution. This can be done easily by using the dist_copy command, e.g.:

```
dist_copy lib $PROJROOT/a/x/y/$ARCH/some_obj.so
dist_copy doc $PROJROOT/a/b/doc/README.txt
```

The dist_copy command creates targets to copy files if they are newer than the ones currently in the distribution. The first argument to dist_copy is the target directory, relative to the distribution directory. If the directory does not exist yet, it will be created (recursively).

The specified executable files will be copied and renamed. The variable $DIST_RENAME controls how this new name is determined, the default value is {${ITEM}.orig}. $ITEM will later expand to the original name of the executable. Next, a wrapper shell script will be created by the name of $DIST_WRAPPER. For each executable, a symbolic link is created, named as the original executable and pointing to the wrapper. This means that calling the link calls the wrapper, and the wrapper then calls the "real" executable.

This mechanism allows for setting an arbitrary number of environment variables (in the wrapper) before calling the executable. You can simply add shell script command lines to the wrapper using the dist_script command, e.g.:

```
dist_script "export SOME_PATH=\$DIST_PATH/lib"
```

The quoted shell command line will set the shell variable $SOME_PATH prior to calling the executable. Within the script, the shell variable $DIST_PATH points to the directory where the distribution is currently located (which is determined by the wrapper script). This

mechanism is also used to set the runtime shared library search path so that libraries are first looked up in the distribution's library directory.

This means that as long as your software does not contain any hard-coded path names or similar things, the complete distribution can be relocated everywhere by simply moving it. So if you need to work with search paths or similar things within your programs, you should take care that those paths can be set via environment variables. If that is the case, you can always set the paths to reasonable and relocatable defaults in the wrapper script.

4.9 `db`: Simple Database Interface

Many automation tasks rely on some sort of database query. Therefore, `tmk` comes with a small textfile-based database interface which can be used for managing small amounts of data very conveniently. Just like the `TMakefile`, `tmk`'s database files are simply TCL source code files. The `db` module provides several new commands.

4.9.1 Database definition

A `tmk` database is a list of data records. A record is a list of fields. A field is a pair consisting of the fieldname and a value. Here is an example of how to define one such record, representing one instance of an address:

```
db_record {
    field lastname  "Mustermann"
    field firstname "Erika"
    field street    "Bahnhofstr.~1"
    field zipcode   "54321"
    field city      "Irgendwo"
    field phone     "+49 12 3456789"
}
```

A file containing a collection of such definitions is called a database file. Let's assume you have defined some records/addresses in the file `myaddr.db`. The `tmk` command `db_read` is used to create a database (a list of records in the main memory) from the file:

```
module db
set db [db_read "myaddr.db"]
```

The second line of code will simply execute the specified database file, and the `db_record` and `field` commands will append fields and records to the list that will then be stored in the variable `$db` that was specified in the example.

The resulting database is simply a list of records, which each record being again a list of fields. Each field is a pair consisting of the field name and the field's value.

4.9.2 Working on the Database

You can operate on the list of records with all standard TCL and `tmk` commands. For more convenience, there are some database commands for working on lists of records or on single records.

In order to generate the right selection and order of records, there are two fundamental database commands.

`db_select` <*database*> <*match-expr*>

This command is used for selecting a number of records from the specified list (database). It returns a new database. <*match-expr*> is some valid TCL expression which may contain field names as variable names. This means `tmk` will go through all records and, before evaluating <*match-expr*> for the current record, create one variable for each valid field in the current record, with the variable name equal to the field name. Example:

```
set thecity "Irgendwo"
set people_in_irgendwo [db_select $db {$city == $thecity}]
```

This line of code selects all records from the previous example in which the `city` field contains the value `Irgendwo`. Please note that again, the expression is evaluated in the context of the caller, so that all currently visible variables can be used.

Instead of this very simple matching expression, you can use arbitrarily complex TCL/`tmk` expressions. The simplest expressions are `"1"` for selecting all records, and `"0"` for selecting no records.

The second fundamental command working on a list of records is used for sorting the records. It has a similar form:

`db_sort` <*database*> <*sort-expr*> [<*order*>]

This command takes a database, sorts it according to a key which is determined by the specified <*sort-expr*>, and returns a new database. The <*sort-expr*> is again some TCL/`tmk` expression containing field variables. The result of this expression is used for comparing the record to others (using ASCII comparison). The optional <*order*> argument defines whether the sorting will be done in ascending or descending order (`"inc"` (default) or `"dec"`). Here is an example:

```
set sorted_db [db_select $db {"$lastname__$firstname__$city"}]
```

This example sorts the database by the last names containes in the records. For records with the same last names, it will the consider the first names, and finally the city.

Please note that for the field-variable expressions used in the examples above, the corresponding fields must be defined in each record. Techniques to make sure that certain fields are always defined are discussed in the section about *Advanced Record Definitions*.

4.9.3 Working on Single Records

After selecting and sorting the database, you obtain a list of records. Again, you can operate on this list with the standard TCL/`tmk` commands in order to do something with each record or some of the records. After you obtain a single record from the list, the most basic command to do something with that record is

`db_with_record_do` <*record*> <*command*>

This command allows you to execute any TCL/`tmk` code 'on' the specified record. This means that, like for the matching and sorting expressions mentioned in the previous section, `tmk` sets one variable for each field of the record, with the variable name matching the field name. Example:

```
foreach rec $sorted_db {
    set last_city "<undefined>"
    db_with_record_do $rec {
        if { ($last_city != "<undefined>") && \
             ($city != $last_city) } {
            puts stderr "---"
        }
        puts stderr "$lastname, $firstname, $city"
        set last_city $city
    }
}
```

This example goes through all records of `$sorted_db` and outputs a line containing name and city in the record. In addition to that, it separates records from different cities by a short line.

If you want to create new database files from old ones, you can do this by simply writing a new file and then writing records into that file using the `db_output_record` command, e.g.:

```
set f [open "sorted_addresses.db" w]
puts $f "# this file has been generated automatically"
foreach rec $sorted_db {
    puts $f [db_output_record $rec]
}
close $f
```

The `db_output_record` command will write the record in the form in which it is needed for the `db_read` command. For printing a record on the screen (to be read by a user) you may use `db_format_record`, which will print the record in a more readable form.

If you want to access a single field of a single record, you may use another command for convenience:

`db_value` <*record*> <*fieldname*>

returns the value of the specified field in the specified record.

4.9.4 Advanced Record Definitions

So far, we have relied on the assumption that all kinds of fields needed in our sorting and selection expressions are actually defined in each record. Since it is not easy to ensure this, it might be more convenient to define default values for some fields so that there won't be any problem if some record does not define the field. Also, you may want to make use of some common definitions etc., and you may want to construct "compound" fields automatically which combine the contents of several simple fields. There are two constructs for making all this possible:

`db_record_header` <*script*>

`db_record_cons` <*script*>

The "header" script will be executed at the *beginning* of each record definition, and the "constructor" will be executed at the *end*, after parsing all the commands in a `db_record` procedure. So you can define default field values as well as shortcuts in the header, and construct compount fields and perform integrity checks in the constructor. Every 'field' command will cause the corresponding local variable to be set to the value of the field. Here is an example:

```
db_record_header {
    field firstname ""
    field lastname ""
    field street ""
}

db_record_cons {
    field name "$lastname, $firstname"
    if {![info exists city]} {
       set rec [db_format_record $therecord]
       __ExitErr "must specify a city in $rec"
    }

}
```

As you can see in the above example, the variable `$therecord` is reserved and contains all fields defined so far for the current record. You can output it via the functions `db_format_record` or `db_output_record`.

4.10 `latex`: Using LaTeX, BibTeX etc. *(experimental!)*

The LaTeX module is thought for automating the task of compiling LaTeXsource files into DVI, Postscript, or PDF documents. In order to do so, the module has to generate several types of dependencies and call `latex` a number of times in order to make sure that all references etc. are properly resolved. The module accomplishes this by doing the following:

- Detect all "main" LaTeXfiles by looking for files matching `*.tex` and searching for the code `\begin{document}` (with some whitespace characters allowed in between).
- Generate dependencies to included LaTeX files by parsing `\input` statements.
- Generate dependencies to bibliography databases by parsing `\bibliography` statements.
- Analyse the output of a `latex` invocation in order to determine if there remain unresolved references or citations. If unresolved references occur, run `latex` again. For unresolved citations, if the used bibliography databases are newer than the LaTeXsource, run `bibtex` and then run `latex` again twice.

The parsing mechanism used for `\input` and `\bibliography` statements is very limited. For example, it cannot resolve macros or similar things. Also, it will only look for files in the paths specified in the `tmk` variables `$TEXINPUTS` and `$BIBINPUTS`. If the corresponding environment variable is set, `$TEXINPUTS` is set to `$env(TEXINPUTS)`, and the path separators ":" are replaced by whitespace in order to obtain a TCL list. If the environment variable is not set, `$TEXINPUTS` defaults to `"."`. The same applies to `$BIBINPUTS`.

The module defines rules for compiling `.tex` into `.dvi`, `.dvi` into `.ps`, and `.ps` into `.pdf`. Corresponding to the values of `$USE_AUTO_DVI`, `$USE_AUTO_PS`, and `$USE_AUTO_PDF`, the LaTeX sources will be compiled into DVI, Postscript, and PDF documents. All targets are generated in the local directory, not in the `$ARCH` directory (even if an empty `$ARCH` directory will be created).

The used programs are determined by the variables `$LATEX`, `$BIBTEX`, `$DVIPS`, and `$PSTOPDF`. Flags can be set via `$LATEXFLAGS`, `$BIBTEXFLAGS`, `$DVIPSFLAGS`, and `$PSTOPDFFLAGS`.

When calling `tmk clean`, the variable `$LATEX_CLEAN_SUFFIXES` determines the files to be deleted. The default is

```
*.dvi *.aux *.log *.bbl *.blg *.toc
```

The variables `$LATEX_UNDEF_REF` and `$LATEX_LABELS_CHANGED` determine which messages will cause the module to rerun `latex` or call `bibtex`.

4.11 Writing Your Own Modules

Sorry, no useful tips yet.

Chapter 5

Installation and Configuration

5.1 Installing `tmk` on your system

Installing `tmk` is very easy. Just copy the files into your favourite directory. The `tmk` source tree consists of three directories:

- `src/`: the `tmk` script source file
- `modules/`: the module script source code
- `doc/`: the documentation TeX/dvi files

To install `tmk` on your system, just change the code line setting the variable `$__TmkDir` in the `tmk` script, and include the `src/` directory in your shell's `$PATH` variable.

5.2 Configuring tmk

Appendix A

Misc `tmk` Functions and Variables

A.1 List Operations

Since most of the operations in `tmk` modules and `TMakefile`'s deal with lists of file names, it is useful to have some basic list operations at hand.

- `lindex` <*list*> <*n*>: picks the n'th item from a list [std. TCL]
- `lappend` <*listname*> <*elements ...*>: appends any number of elements to a list variable
- `lrange` <*list*> <*n1*> <*n2*>: creates the sublist including all elements from index <*n1*> to <*n2*>. May use "end" as placeholder for the last element of the list. [std. TCL]
- `lfilter` <*list*> <*T-expr*>: copies into a new list all those items for which the T-expression is true. T-expressions are explained below. Example:

```
set x [lfilter "a ab c d e" {[string match a* $ITEM]}]
-> x = "a ab"
```

- `lmap` <*list*> <*T-expr*>: creates a new list, containing the results of evaluating the expression for every item of the list and concatenating it to the return value list. T-expressions are explained below. Lists are decomposed once so that it is very easy *not* to generate lists of lists. Example:

```
set x [lmap "a b c" {$ITEM $ITEM}]
-> x = "a a b b c c"
set x [lmap "a b c" {[list [list $ITEM $ITEM]]}]
-> x = "{a a} {b b} {c c}"
```

- `lminus` <*list1*> <*list2*>: subtracts <*list2*> from <*list1*>. Example:

```
set x [lminus "a b c d d e e" "x y d"]
-> x = "a b c e e"
```

- `lremove` <*list*> <*pattern*>: remove all elements from <*list*> which match <*pattern*>. Example:

```
set x "ax bx by ay"
lremove x "a*"
-> x = "bx by"
```

- `lcontains` <*list*> <*elem*> tells whether <*elem*> is contained in <*list*>. Example:

```
set x [lcontains "a b c d d e e" "a"]
-> x = 1
```

The above-mentioned T-expressions are similar to those you can use within a `target` statement, except that the special variable names are a little bit different, and the expression is evaluated in the context of the caller of the list operation. You may use the following special variables within such a T-expression:

- `$ITEM`: the full name of a single list element
- `$IROOT`: all characters of `$ITEM` up to (but not including) the last dot (cf. TCL command `[file rootname $ITEM]`)
- `$IEXT`: all characters from the last dot on, or the empty string if name contains no dot (cf. TCL `[file extension $ITEM]`)
- `$IDIR`: all characters up to the last slash, or '.' if there are no slashes in the name (cf. TCL `[file dirname $ITEM]`)
- `$ITAIL`: all characters after the last slash, or the empty string (cf. TCL `[file tail $ITEM]`)
- `$IBASE`: all characters after the last slash and before the last dot (cf. `[file rootname [file tail $ITEM]]`)

A.2 Execution, Logging, and Debugging

- `assert` <*condition*>: evaluates the condition on the current level and exits with an error if it does not evaluate to 1.

- `cmd` <*args*>: performs a `[eval exec` <*args*>`]` at the global execution context (`uplevel #0`). This implies that the first of the arguments will be executed as a command in a shell, using all further arguments as its parameters values. The additional `[eval]` statement sees to that one single list of arguments will be generated, even if <*args*> contains one level of nested lists like `$CCFLAGS`. `cmd` will also catch system errors, print the error message, and exit `tmk`. Standard input, output, and error are piped to the terminal.

- `__Log` <*msg*>: outputs <*msg*> to the standard error channel. This is used for all important messages which the user should see even when not debugging. Log messages can be turned off with the `-silent` command line option.

- `__Dbg` <*msg*>: outputs <*msg*> to the standard error channel. This is used for all those messages which can be helpful for debugging, but should not be printed by default. Debugging messages can be turned on with the `-debug` command line option.

- `__ExitErr` <*msg*>: outputs <*msg*> to the standard error channel and terminates `tmk`. If debugging is switched on, also shows a stack trace at exit time.

- `SetIfUndef` <*varname*> <*value*>: set variable to value if it is not defined already.

- `beforeBuilding` <*TCL script*>: In order to execute commands *after* all variables have been set in the `TMakefile`, the modules make use of the `beforeBuilding` command. Sometimes it can be reasonable to use it directly in the `TMakefile`, too. The command appends the specified commands to a list. After having parsed the whole `TMakefile`, just before starting to build the first target, `tmk` will process each element of that list (by means of a TCL `[eval]` statement) in the order of their appearance. This provides a means of performing additional checks and computations after all user variables have been set to their final values. This is used heavily in most of the modules.

A.3 Target Names, File Names, Directories

- `shortTargetName` <*target*>: returns the short form of the specified target name. This is done by removing `$ARCH` if it is the last part of the path before the filename (and if `$USE_ARCH` is set), and by removing trailing `./` paths. The command is for example useful when specifying additional dependencies via the `depend` command and the target name comes from some external program like makedepend.

- `fullTargetName` <*target*>: returns the complete target path and filename. If `$USE_ARCH` is set, and if the last element of the path is not `$ARCH`, it will be added

before the filename. This is useful if the target file has to be passed to some external program outside of any `target` command.

- `PathIsAbsolute` <*path*>: tells whether the specified path is an absolute path or not. Depending on the operating system you're on, this can be hard to tell. On UNIX systems, only absolute paths start with a "/".
- `NormalizeFilename` <*varname*>: normalizes the filename contained in the specified variable. Changes the variable and returns the resulting value. Normalizing means making filenames comparable. For example, collapse `A/B/../C` to `A/C` where possible.
- `__CreateDirRecursively` <*dir*>: create a directory and all parent directories which do not already exist
- `SearchInPathlist` <*filepattern*> <*pathlist*>: for all paths in the pathlist, return all files which match the pattern `path/$filepattern`.

Appendix B

Index of Variables

This section lists all environment and global variables which are used by the `tmk` core or by any of the standard modules. The table also lists the section where more detail about the function of the variable can be found, and cross links to other variables or commands which are used in the same context.

Please note that all variable names starting with a double underscore (__) are reserved for the `tmk` core system.

Global Variables

Name	Meaning	Sec.
`$ARCH`	name of the currently active architecture; see also: `$ARCH_BASE`, `$USE_ARCH`	3.8
`$ARGS`	arguments to be passed to a `tmk` subprocess, usually the same arguments as have been passed to the current `tmk` process.; see also: `subdir` command, `$TMK`	3.9
`$AUTODEPEND`	list of automatically generated targets for reg-generating dependency files; see also: `$AUTOTARGET`, `$EXCLUDE`	4.1.9
`$AUTOTARGETS`	list of automatically generated targets; this variable is modified by several modules; after parsing the `TMakefile`, the `$EXCLUDE` targets are removed from this list and the remaning targets get built.; see also: `$MAKE_AUTOTARGETS`, `$AUTODEPEND`, `$EXCLUDE`	4.1.1
`$C`	command to be executed for compiling C code; see also: `$CFLAGS`	4.2
`$CC`	command to be executed for compiling C++ code; see also: `$CCFLAGS`	4.2

$CCEXTENSIONS	list of file extensions which are supposed to mark C++ files; these are used to generate .o targets for all detected C++ files; must be specified *before* calling the CC module; see also: $CEXTENSIONS	4.2
$CCFLAGS	list of flags to be passed to the C++ compiler; see also: $CC, $CFLAGS	4.2
$CEXTENSIONS	list of file extensions which are supposed to mark C files; these are used to generate .o targets for all detected C files; must be specified *before* calling the CC module; see also: $CCEXTENSIONS	4.2
$CFLAGS	list of flags to be passed to the C compiler; see also: $C, $CCFLAGS	4.2
$CLEAN_PATTERNS	name pattern list for files which should be deleted when calling tmk clean; additionally, the $ARCH directory will be deleted if $USE_ARCH is on; see also: target clean, $USE_ARCH	4.1.8
$CURRENTDIR	after parsing the TMakefile: current directory relative to $PROJDIR; see also: $PROJDIR, $DIRTAIL, default module	3.10
$DEPEND_EXCLUDE	list of directory patterns; dependency files matching one of the patterns will not be checked for time-stamp or existence during building.; see also: depend command	3.5
$DIRTAIL	after parsing the TMakefile: tail of the current directory (short directory name), also used as base name for the automatic local library; see also: $CURRENTDIR, default module	4.1.4
$DOXYFILE	name of the patched/final Doxygen configuration file.; see also: $DOXYFILE_INPUT, $DXX_*, doxygen module	4.6
$DOXYFILE_INPUT	name of the input configuration file to be generated by Doxygen which is to be patched by tmk; see also: $DOXYFILE, $DXX_*, doxygen module	4.6
$DOXYGEN	path/name of the doxygen executable; see also: doxygen module	4.6
$DXX_*	all variables starting with DXX_ correspond to a Doxygen config variable (without the DXX_ prefix); the variables defined in the TMakefile will be written into the patched Doxygen config file $DOXYFILE; see also: $DOXYFILE, $DOXYFILE_INPUT, doxygen module	4.6
$EXCLUDE	list of targets to be excluded from the automatic target generation process; see also: $AUTOTARGETS	4.1.1
$LD	name of the linker command; see also: $LDFLAGS, $LIBPATH, $SYSLIBS, $PROJLIBS	4.1

$LDFLAGS	global flags for linking; see also: $LDPROC, $LIBPATH, $SYSLIBS, $PROJLIBS	4.1
$LEX	program to be executed for generating code from lexicographical description files.; see also: $LEXFLAGS, $LEXSUFFIX,$LEXSUFFIX_C	4.5
$LEXFLAGS	flags for $LEX; see also: $LEX, $LEXSUFFIX,$LEXSUFFIX_C	4.5
$LEXSUFFIX	suffix for the lexicographical description file (default is l). Must be set before calling the lex module.; see also: $LEX, $LEXFLAGS, $LEXSUFFIX_C	4.5
$LEXSUFFIX_C	suffix for the code file to be generated from a lex file. Must be set before calling the lex module.; see also: $LEX, $LEXFLAGS, $LEXSUFFIX	4.5
$LIBPATH	list of additional library paths, will be used for generating -L and -rpath options for the linker; see also: $LD, $LDFLAGS, $SYSLIBS	4.1.2
$LIB_OBJ	after TMakefile parsing: list of .o files to be included in the automatically generated library; consists of all .o $AUTOTARGETS, excluding all $EXCLUDE targets and all object files in $PROG_OBJ_FILES; see also: $USE_AUTO_LIB, $PROG_OBJ_FILES	4.1.4
$LINK_LIB_TWICE	switch on/off the duplicate linking of all libraries.; see also: $LDPROC, $LDFLAGS, $LIBPATH, $SYSLIBS, $PROJLIBS, $LIB_LOCATIONS	4.1.2
$MAKE_AUTOTARGETS	switch on/off the appending of auto-detected targets to the $AUTOTARGETS variable; see also: $AUTOTARGETS	4.1.1
$MAKE_PROGRAMS	switch on/off (1/0) the automatic linking of executable programs; see also: $PROGRAMS, $LD, $LDFLAGS	4.1.5
$MAKE_STATIC_LIB	switch on/off (1/0) the automatic creation of a static library for each processed directory; the library will contain all automatically generated object files except for those corresponding to executable programs; see also: $USE_AUTO_PROGRAMS, $PROGRAMS, $AUTOTARGETS	4.1.4
$MAKE_SHARED_LIB	like $USE_AUTO_LIB, but for a shared library; see also: $USE_AUTO_LIB	4.1.4
$MOCFLAGS	flags for the QT meta object compiler, “moc”; see also: $QTMOC, $QTDIR, $QTPATTERNS	4.3
$MODULES	list of all modules called so far; see also: module, $env(TMK_MODULE_PATH)	4

$PROGRAMS	list of executables to be built; tmk will try to link the corresponding .o files with the specified system and project libraries in order to generate executables; see also: $LD, $LDFLAGS, $USE_AUTO_PROGRAMS	4.1.5
$PROG_OBJ	after TMakefile parsing: list of all object files corresponding to an executable specified by $PROGRAMS; see also: $MAKE_PROGRAMS, $PROGRAMS	4.1.4
$PROJDIR	location of the project directory and the project makefile TMakefile.proj. If no project makefile exists, defaults to the current directory; see also: $PROJLIBS, -proj and -noproj command line options	3.10
$PROJLIBS	list of libraries from the current project to be linked; a library is specified by the project directory it is in (e.g. a/b/X corresponds to $PROJDIR/a/b/X/$ARCH/libX.a); see also: $PROJDIR, $LD, $LDFLAGS, $LIBPATH, $SYSLIBS	3.10
$PROJROOT	after parsing the TMakefile: parent directory of the current project directory ($PROJDIR/..); see also: $PROJDIR, $PROJLIBS, -proj and -noproj command line options	3.10
$QTDIR	directory where the QT package is installed; used to infer the location of QT libraries, include files, and the meta object compiler; see also: $QTMOC, $MOCFLAGS, $QTPATTERNS	4.3
$QTMOC	command to be executed in order to run the QT meta object compiler "moc". Default is $QTDIR/bin/moc; see also: $MOCFLAGS, $QTDIR, $QTPATTERNS	4.3
$QTPATTERNS	list of file patterns which refer to QT header files. These files will be precompiled by moc into the corresponding .moc.C files, and the .moc.o object files will be added to the automatic targets.; see also: $QTMOC, $QTPATTERNS	4.3
$SUBDIR_EXCLUDE	list of directory names which will not be considered for automatic subdirectory processing; the current $ARCH/ will be added automatically.; see also: subdir command, -local command line option	3.9
$SYSLIBS	list containing either single library names (short form, e.g. m for the math library), or sublists of the form {path lib1 lib2 lib3...} for direct library path assignment; see also: $LDPROC, $LDFLAGS, $LIBPATH, $LIB_LOCATIONS	4.1.2

$TARGETS	list of all current toplevel targets (either specified by build commands or at the tmk command line; see also: build command, tmk command line options	3.1
$TMK	the command to be called for recursive subdirectory processing. Usually contains the path to the tmk executable of the current process.; see also: subdir command, $ARGS	3.9
$USE_ARCH	switch on/off (1/0) multiple architecture support; see also: $ARCH	3.8
$YACC	program to be executed for parser generation from grammars; see also: $YACCFLAGS, $YACCSUFFIX, $YACCSUFFIX_C, $YACCSUFFIX_H	4.4
$YACCFLAGS	flags for $YACC; see also: $YACC, $YACCSUFFIX, $YACCSUFFIX_C, $YACCSUFFIX_H	4.4
$YACCSUFFIX	suffix of grammar files. Default is y. Must be set before calling the yacc module.; see also: $YACC, $YACCFLAGS, $YACCSUFFIX_C, $YACCSUFFIX_H	4.4
$YACCSUFFIX_H	suffix of the header files generated from grammar files. Default is hh. Must be set before calling the yacc module.; see also: $YACC, $YACCFLAGS, $YACCSUFFIX, $YACCSUFFIX_C	4.4
$YACCSUFFIX_C	suffix of the code files generated from grammar files. Default is C. Must be set before calling the yacc module.; see also: $YACC, $YACCFLAGS, $YACCSUFFIX, $YACCSUFFIX_H	4.4

Environment Variables

Name	Meaning	Sec.
$env(HOME)	user's home directory; this is used to create a .tmk directory for configuration cache files etc; see also: tmk command, configuration	5
$env(TMK_HOME)	directory in which the tmk system resides; see also: tmk command, configuration	5
$env(TMK_MODULE_DIR)	list of paths where to look for tmk modules; see also: module command	4
$env(TMK_TCLSH)	TCL shell program to be used for executing the tmk scripts; see also: tmk command, configuration	5

Appendix C

Index of Built-In Functions

Please note that all function names starting with a double underscore (__) are reserved for the `tmk` core system.

Appendix D

Index of `tmk` Command Line Options

The syntax for calling `tmk` is as follows:

> `tmk` <*options ...*> <*targets...*>

After listing the desired options, the user may explicitly specify any number of targets to be built. If no targets are given, `tmk` will try to build all targets specified via the `build` command in the `TMakefile`. Options are always preceeded by a '-'. The available options can be listed by invoking `tmk -help`:

- `-help`: output short message explaining command syntax
- `-prf`, `-dbg`, `-std`, `-opt`, `-max`: select the code level from profiling (prf) up to maximal optimization (opt)
- `-f file`: use 'file' instead of `TMakefile`
- `-proj file`: use 'file' as project makefile [default: search upwards in the parent directories for `TMakefile.proj`]
- `-noproj`: do not search for any project makefile
- `-priv file`: use 'file' as private project makefile [default: search upwards in the parent directories for `TMakefile.priv`]
- `-nopriv`: do not search for any private project makefile
- `-local`: skip subdirectory processing
- `-force`: build all specified targets unconditionally (meaning even if they do not need to be updated).
- `-debug` / `-nodebug`: toggle debugging output on/off

- `-silent` / `-verbose`: toggle logging output on/off
- `-mfdepend`: include the `TMakefile` as dependency for every target to be built, so if the `TMakefile` has changed, all targets will be rebuilt. If you want this to be permanent, just set the global variable `$__SelfDepend` to 1.

There are some more options, which are not used normally. Please use with care.

- `-reconfig`: causes `tmk` to re-generate all config files for the currently active system
- `-arch name`: set `$USE_ARCH` to 1 and `$ARCH` to 'name'. This overrides the name of the architecture, which is normally set automatically according to the different components of your system and to the selected code level.
- `-rules`: output the rule database instead of building the targets
- `-cmd 'script'`: execute the TCL script prior to reading the default module and parsing the `TMakefile`
- `-prefix 'string'`: print the specified string before every line of output [default is `tmk:`]. This is used internally by `tmk`, e.g. for subdirectory processing