installing & configuring

Hartmut Schirmacher
Max-Planck-Institut für Informatik

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

Installing & Configuring tmk

Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85, 66123 Saarbrücken

[draft version, August 4, 2000]
available from www.tmk-site.org

Contents

Introduction

This document provides you with the necessary information for configuring TMK according to your needs. Section 1 is a quick reference for getting TMK running on your system using mostly default configuration. The later sections go into the details of the configuration system, starting with the overall configuration design and going into the details of three major parts: architecture-dependent configuration, site-dependent configuration, and finally the meta compiler model and compiler configuration.

If you want to do more than just standard configuration tasks, and you are not familiar with the TCL language, please consider reading the first chapter of the TMK tutorial or any other introduction to the TCL language first.

If you have carefully read this document, and still have serious problems configuring or running TMK, please do not hesitate to contact the TMK developer team through the TMK web page `www.tmk-site.org`.

1 Getting TMK Up and Running

This section briefly explains the steps that are needed to install TMK from scratch and get the basic system running.

1.1 Unpack files from the archive

When you obtain the TMK distribution from the TMK web site, it comes as a packed archive file `tmk-`*`release-number`*`.tar.gz`. You have to extract the files from these archives in some software package directory, e.g. with the `gunzip` and `tar` tools on UNIX systems:

```
gunzip tmk-release-number.tar.gz
tar xvf tmk-release-number.tar
```

or programs like `winzip` on Windows systems (using its graphical user interface). The extracted files will reside in a subdirectory called `tmk-`*`release-number`*. The actual TMK system is located in `tmk-`*`release-number`*`/tmk`. We will call this directory the TMK *home directory*.

1.2 Install TCL

Next, you need to make sure that you have a reasonably new version of TCL installed on your system. The current TMK release requires TCL versions 8.x, since TMK relies on the namespace functionality and operations such as `file copy`. You can obtain TCL for free from `http://dev.scriptics.com`. Usually, you should have a program called `tclsh` in you path. You can find out its version by starting `tclsh` and then typing

```
info tclversion
```

You can leave the TCL shell again by typing `exit`. If the TCL shell used for TMK is not the one in your system's search path, you can set the environment variable `TMK_TCLSH` to the desired shell program, and TMK will start using that shell.

1.3 Make TMK accessible

Next, you should include TMK's `src` subdirectory in the system's search path (`PATH` environment variable), or create a symbolic link or alias to the the actual 'executable'[1]. On UNIX systems, this is a TCL script simply called `tmk` in the `src` subdirectory,

[1] Don't forget to type `rehash` if you're using `csh` or `tcsh` ...

while on Windows system this script is invoked via a batch file called `tmk.bat` in the same directory. Furthermore, on Windows systems you must set the environment variable `TMK_HOME` to the absolute location (also including the drive letter) of the TMK home directory, so that the batch file can find the TCL script.

If it should happen that TMK cannot find its own location on UNIX systems (e.g. it crashes inside the `::tmk::scriptname` function), you can also try to set the `TMK_HOME` environment variable manually. If even this does not work, please report the problem via `www.tmk-site.org`.

1.4 Site-Specific Configuration

Now the basic TMK system should be ready to run. The only thing left to do is configuring the different TMK modules you want to use, mostly by specifying a number of site-dependent path names. Let's start by typing

```
tmk -sysinfo
```

Now TMK should display some information about the system it is running on. Particularly interesting items are the name of your machine (HOST), the currently used network domain (DOMAIN), and the operating system class (OSCLASS). If you're not in a network, the domain name will be set to `localdomain`.

Now change to `config/site` in your TMK home directory. This is the location where TMK tries to find *site*-specific configuration. On your system, TMK will look for the following files, and read all existent files in the specified order:

- `site-config.tmk`
- `DOMAIN`
- `DOMAIN:OSCLASS`
- `HOST:DOMAIN`
- `HOST:DOMAIN:OSCLASS`

where `HOST`, `DOMAIN`, and `OSCLASS` are again the placeholders for the actual system information. So now you can create one or more of these files in order to do the site-specific configuration for your machine or all machines in your network. For example, if your machine is called `mymachine`, and you are not connected to a network, you could create the file

```
mymachine:localdomain
```

e.g. by copying one of the example configuration files which are called `sample.*` in the same directory. These sample files are named in such way that you can guess for which kind of systems they are designed. Here is a small part of such an example site config file for a Linux machine:

```
# example from site config for a RedHat Linux system

# specify include/lib path + lib names for X11
config set x11::INCPATH /usr/X11R6/include
config set x11::LIBPATH /usr/X11R6/lib
config set x11::LIBS    {Xt Xi Xext Xmu X11}

# specify include/lib path + lib names for QT
config set qt::INCPATH   /usr/include/qt2
config set qt::LIBPATH   /usr/lib/qt2
config set qt::LIBS      {qt}
# on UNIX, for QT you also need X11
config set qt::DEPEND     {x11}
```

As you can easily derive from this example, the main purpose of the site-specific configuration is to specify the directory names, library names, and similar things such as the executable files for some helper programs. Just check if the paths in the example files match those on your system, and modify them if necessary. If you want to know more about the meaning of the `config set` command and the config variables specified in the files, please refer to the later sections of this document and other detailed documentation on TMK.

Of course you only need to configure those TMK modules that you intend to use. You can simply comment out all other lines by preceeding them with a hash character ('#'), or you can simply delete the lines from the file.

After completing the site-specific configuration, just type

```
tmk -reconfig
```

and TMK will read all the relevant config files and store the result in its so-called *config cache file* for the current system. The name of that file is also displayed after successful configuration.

1.5 Problems . . . ?

After the steps described above, TMK should be ready to go. However, there can always arise circumstances in which the described procedure fails. If so, please con-

sider looking at the *frequently asked questions (FAQ)*, and specifically the *installation FAQ* in the documentation section of the TMK web pages[2], or contact the TMK team.

2 The Configuration System

One of TMK's main features is the distinction between the actual control files for the building process (e.g. a `TMakefile`) and system-specific configuration settings. There are several benefits from that, e.g.

- system-dependent code is mostly hidden from the user
- control files are portable and transparent
- configuration only needs to be done once, centrally (e.g. in a multi user network environment)
- updates to new software package versions can be done transparently for all users

In order to support all this, TMK has a dedicated subsystem that consists of a number of files defining variables and procedures, depending on the system TMK is actually running on. This section gives an overview of the system, followed by some general information about what config files are made of.

2.1 The Config Cache

As already mentioned, the config system processes a number of directories and files, depending on the system TMK is actually running on. Since this configuration takes some time to be computed, the settings are stored in a so-called *config cache*, so that later TMK only needs to read that single cache file instead of running through the whole configuration process again. The config cache file can consist of centrally defined settings plus user-defined settings, so one cache file is stored per user and system. You can find out which system you're on and which config cache file will be used by typing

```
tmk -sysinfo
```

Furthermore, you can find out which files TMK will process on the current system by typing

```
tmk -reconfig
```

[2] `http://www.tmk-site.org/doc`

This will cause the config files to be re-read and the config cache to be rebuilt, and while doing that TMK displays the names of all the files that are processed. In addition to this explicit cache update, there is a file called `rebuild_cache` in TMK's `config/` directory. Whenever a user starts TMK, and the user's current cache file is older than `rebuild_cache`, TMK will do a `-reconfig` on its own. This way the TMK administrator can 'commit' global changes to all users.

The config cache name contains the name of the machine, the network domain name, the operating system (and version), and some more information. So if you have the same directories mounted under different operating system versions, you will have one cache file for each version. Similarly, you have a different cache file when you are connected to a network than when you are not. This way you can easily account for software package locations or other configuration options that may change depending on the different aspects of your system.

Here is an example: assume an Intel-based PC named `horst` that is usually booted under Linux, but also under Windows. Both systems share the same directory structure for compiling the softare projects. The computer may be, but is not necessarily connected to a network domain named `mydomain.com`. In that case, you may have different cache files likes this:

```
config-i686-pc-linux-2.2-horst-mydomain.com
config-i686-pc-linux-2.2-horst-localdomain
config-i686-pc-mswin-98-horst-mydomain.com
config-i686-pc-mswin-98-horst-localdomain
```

As you can see, this example is for a Linux 2.2 kernel and Windows 98, respectively. On UNIX systems, the cache files are stored in the user's home directory, in `.tmk/cache/`. On Windows systems, the cache is either stored in the registry, or in a user home directory if the `HOME` environment variable is specified.

2.2 Config Structure Overview

The configuration system accounts for three ways in which systems may differ, reflected in the three main branches of the `config` subdirectory[3]:

- system architecture, e.g. operating system (`arch/`)
- software environment, e.g. compiler (`soft/`)
- site-specific configuration, e.g. location of software packages (`site/`)

[3]Note that the absolute location of these three directories can be queried via the TCL variables `$tmk::dir_arch`, `$tmk::dir_soft`, and `$tmk::dir_site`. The compiler config resides in `soft/comp` (`$tmk::dir_comp`).

In the system architecture branch, things such as OS-specific file name conventions are set up. Furthermore, this branch utilizes OS-specific system tools for auto-detecting reasonable default values for most parts of the configuration.

The software environment branch currently only contains compiler configuration. The reason why compilers have their own configuration branch is that compilers are complex tools with lots of parameters and very different capabilities. So TMK internally uses a *meta compiler* model: whereever possible, the TMK user works with the abstract meta compiler, and the compiler configuration maps the meta compiler operations to the actual compiler implementations.

More details about what needs to be configured in each of these parts can be found in Sections 3 – 5 further on in this document.

2.3 Config File Syntax and Commands

All config files are TCL/TMK scripts, meaning that they will be read and interpreted by TMK using the TCL `source` command. So the file syntax is that of TCL. If you want to learn more about the TCL language, you can have a look at the TMK tutorial available from `www.tmk-site.org`, or at one of the many comprehensive TCL tutorials that are commonly available (browse `dev.scriptics.com` for a list of tutorials and books on TCL).

The task of a config file is to register a number of variables and procedures so that they can be stored in the config cache (see previous section). So TMK defines a command for querying and modifying the contents of that cache. This command is called `config`, and has a number of subcommands. From the user's point of view, the most important ones are `config set`, `config proc`, and `config set_later`, and they will be briefly explained here. First, here is some example code:

```
set pckg /opt/mypckg
config set qt::LIBS    {qt}
config set qt::DEPEND  {cxx x11}
config set qt::LIBPATH "$pckg/qt2"
```

In the example you can see both the TCL `set` command and the TMK `config set` command. Both take exactly the same kind of arguments, one variable name and one value for that variable. If the value expression consists of a single word, it can be specified as it is. If the expression is a list of several space-separated elements, if it contains space or tab characters for other reasons, or if you want to exploit your favourite editor's syntax highlighting mode, the expression should be enclosed in

double quotes or curly braces[4].

The TCL `set` command assigns it the given value immediately. The `config set` command also stores the variable in TMK's config cache.

Usually all config variables for a certain TMK *module* are defined within the same *namespace*, which is at the same time the module's name (e.g. `qt` in the above example). Instead of using the so-called *namespace qualifiers* for each variable name, you can also use the `namespace eval` command to evaluate a piece of code in a certain namespace:

```
set pckg /opt/mypckg
namespace eval qt {
  config set LIBS    {qt}
  config set DEPEND  {cxx}
  config set LIBPATH $pckg/qt2
}
```

In a way very similar to `config set`, you can use `config proc` to define a procedure. Like the TCL `proc` command, it takes two arguments: a list of parameter names and the procedure body script.

```
config proc ::filename_obj {shortname} {
  return ${shortname}.o
}
```

In the above example, a procedure in the global namespace is defined that returns the filename for an object file, given the 'short' name for the file. The function argument is passed as a TCL variable named `shortname`, and the function returns the value of that variable, followed by `.o`. As you see, you can substitute the value of a variable by writing either `$variable-name` or `${variable-name}`. By using braces, you can make sure that characters following the expression will not be interpreted as belonging to the variable name.

In TMK, functions are used for determining system-dependent file names, such as for object files, executables, static and shared libraries, and so on. Compiler configuration also mostly works via procedures.

Besides immediately assigning variables their value, TMK also provides a means for delaying the evaluation of the value until the value is actually needed. This is particularly important for values that need to be set dependent on conditions that can

[4]The difference between these two types of quoting is that within double quotes, *variable and command substitution* (e.g. the `$pckg` expression in the last line of the example) still work, whereas an expression in curly braces is not interpreted in any way. Curly braces, however, can be used to create *nested lists*.

only be evaluated after processing the `TMakefile` or in a different branch of the configuration. For that reason, TMK defines the `config set_lazy` command:

```
config set_lazy glut::LIBPATH {
  switch $link::LINKER {
    "cxx::mipspro" {return "/opt/pckg/glut/lib_mipspro"}
    "default"      {return "/opt/pckg/glut/lib"}
  }
}
```

In this example, the value of `$glut::LIBPATH` depends on the atually chosen linker. The variable will be needed when the `glut` module is loaded from within a `TMakefile`. At that moment, TMK will execute the specified code (containing the `switch` statement etc.), and set `$glut::LIBPATH` according to the value returned by that piece of code (which is treated just like the body of a procedure). However, this does not prevent a user or module from changing the `$link::LINKER` variable even later (e.g. by choosing a different compiler), which may then result in the wrong library path. Note that this is one of the reasons why you should always load language modules (c/cxx) *before* library modules.

2.4 Module Macro Variables

Some config variables, the so-called *module macro variables*, always have the same meaning, and will trigger a certain action automatically when the user invokes the corresponding module. In the example from page 9, all used variables are of that kind, so for example the `$LIBPATH` variable will trigger a piece of code that appends the specified path(s) to the linker module's library path variable. Here is a list of all currently defined macro variables:

`$`*`module`*`::INCPATH`: appends the list elements to the C and C++ modules' include paths

`$`*`module`*`::LIBPATH`: appends the list elements to the linker's library path

`$`*`module`*`::LIBS`: appends the list elements to the linker's current set of external/system libraries

`$`*`module`*`::DEPEND`: specifies inter-module dependencies. If module `X` depends on module `Y`, `Y` will always be loaded and execute before `X`

So if you want to make a set of libraries available to the user in a transparent way, just create a module and specify the location and name of the libs in the site config file.

If on a different system the libraries have a different name or location, this will only be accounted for in the config file, not in every `TMakefile` that uses these libraries. For example, if you want to wrap the math library in that way, you could add a line like this:

```
config set math::LIBS m
```

So instead of specifying the math library explicitly in the `TMakefile`, the user can now write something like

```
module math
```

If on some system using the math library requires an additional library path, or the library has a different name, you can just add the corresponding line in the config system. This may not make much sense for the math library, but for example for thread libraries.

3 Architecture-dependent Configuration

. . . sorry, coming soon . . .

4 Site-dependent Configuration

5 Compiler Configuration

Index