Ocera Make System Manual

Manual for Ocera Make System (OMK) version 0.2-178-gacbe564

Table of Contents

1 Overview

OMK is an advanced make system written entirely in GNU make. Compiling software using OMK requires only GNU Make and standard UNIX utilities (`sh`, `sed`, `cmp`, ...) installed. OMK aims to be developer friendly; to use OMK, you do not need to understand (sometimes) cryptic syntax of Makefiles.

You can use OMK on all platforms where you can run GNU Make including Cygwin and MinGW. MS DOS was not tested.

1.1 Why to Use OMK?

Here we list some of OMK features, which we think are important for choosing of a make system.

- Makefile in source directories are usually very **simple**.
- There is only **one** `Makefile.rules` for most of components of a bigger project.
- OMK greatly simplifies compilation of projects, where source files are spread between **multiple directories**.
- OMK handles properly **dependencies** of source files and libraries, so it is not necessary to recompile the whole project if only several files changed.
- OMK allows to freely **move** cross-dependant components **in directory structure** without the need to update users of moved component. I hate something like `-I../../sched/rtlshwq/include` in makefiles for example. If a component is renamed or version is added to the name, many Makefiles in the project would require an update.
- The above feature is very helpful in **combining components** (libraries) from different projects/developers to a single project by simply creating symbolic links to external components.
- Compilation of an OMK based projects don't require to install any files before successful finish of build.
- OMK allows to call `make` for a particular subdirectory in the source tree.
- Under OMK all products of compilation are stored **out of source directories**. This simplifies work with version control systems and helps when simultaneous compilation for multiple targets/platforms is needed.

1.2 Quick Start

If you get some sources, which are distributed with OMK, usually the following commands are sufficient to compile the whole project.

```
make default-config
make
```

To use OMK in your own project, follow these steps:

1. The newest version of OMK can be found at `http://rtime.felk.cvut.cz/omk/`.
2. Take appropriate `Makefile.rules` (see Section 2.10 [Properties of Specific Makefile.rules], page 14), put it together with leaf `Makefile` to the root directory of your project.

3. Create `Makefile.omk` files in all directories you want to compile something. Please refer to Chapter 2 [User's Manual], page 3 to learn what to write in `Makefile.omk` files.
4. Run `make omkize` in the root directory.

Your project is now ready to compile.

1.3 History

OMK was originally written by Pavel Píša as a solution to have one common make system for OCERA project, where we needed to compile user-space programs, Linux kernel modules and RT Linux modules in one package. Although this system was not accepted for the whole OCERA project. Several individual developers (mostly from Czech Technical University) liked it and started to use it.

As a number of projects using OMK grew it was necessary to modularize the make system to support more "targets". Michal Sojka took care about the process of modularization.

2 User's Manual

2.1 Basic Concepts

The main concept of OMK is very simple. In the heart of OMK are two files `Makefile.rules` and `Makefile.omk`. The former one resides in project root directory and contains all compilation rules needed for compilation of the particular project. There are different `Makefile.rules` for different platforms (Unix, RTEMS, system-less, ...). `Makefile.omk` is stored in every (sub)directory and describes what should `make` perform in that directory (e.g. compile a program from several source files). It uses declarative syntax (assign values to variables) similar to Automake files `Makefile.am`. The content of `Makefile.omk` is described in the following sections.

Since `make` searches by default for a `Makefile` and not for `Makefile.rules` or `Makefile.omk`, there must[1] be a small generic `Makefile` in every directory, whose task is only to find `Makefile.rules` in the actual or any parent directory and include it. This search is performed only once at the beginning of compilation.

The compilation process itself is comprised of several *passes*. Every pass traverses the whole directory structure[2] and does a particular task in every directory of the project. Typically, these passes are:

include-pass
: This pass takes all include files marked for "export" and copies (or links) them to the `include` directory under `_compiled` directory. See Section 2.4.1 [Header Files], page 6.

  Also, during this pass, automatically generated header file are generated according to the current configuration. See Section 2.8 [Configuration and Conditional Compilation], page 9.

library-pass
: During this pass, all include files are in place, so all libraries can be compiled.

binary-pass
: Finally, programs can be compiled and linked against libraries created in the previous pass.

The results of compilation are stored under the `_compiled` directory. This directory is structured as a classical Unix file-system (it contains directories like `bin`, `lib` and `include`) and can be directly copied to the target device or to some directory on a host computer (e.g. `/usr/local`).

Besides `_compiled` directory, there in a `_build` directory. Under this directory are stored some temporary files and intermediate compilation products (object files, dependency files etc.).

---

[1] When USE_LEAF_MAKEFILES is set to '`n`', this `Makefile` can be omitted in subdirectories. See [USE_LEAF_MAKEFILES], page 12.

[2] In future, we are planning some optimization that allows OMK to traverse the directories only once and thus decrease compilation time.

In the next section, we provide an overview of methods, how to invoke OMK from command line. Section Section 2.12 [Interfacing OMK to popular IDEs], page 15 covers running of OMK from popular IDEs.

Sections Section 2.3 [Compiling Programs], page 5 through Section 2.8 [Configuration and Conditional Compilation], page 9 deals with the content of `Makefile.omk`. Its syntax in usual cases compatible to GNU Automake's `Makefile.am` syntax. Also, the scheme for naming variables was inspired by Automake so most OMK variables have the name like '*target_TYPE*'.

2.2 Invoking OMK

Before using OMK for the first time, you have to call:

```
make default-config
```

See Section 2.8 [Configuration and Conditional Compilation], page 9 for details. If you forget to do this, OMK will notice you.

To compile the whole project or only some subtree of the project, call

```
make
```

in the appropriate directory.

To clean files in `_build` directory but not in `_compiled` one, use:

```
make clean
```

To clean the compilation completely, you can either remove `_compiled` and `_build` directories manually, or call

```
make distclean
```

which does the same. This command removes these directories even if you call it from a subdirectory.

To debug compilation problems, you can use `V` variable (see [V], page 4):

```
make V=1
```

You can also set values of some other variables on command line for temporary change something. The example below compiles the code temporarily with debugging information:

```
make CFLAGS="-g -O0 -Wall"
```

If your project uses an alternative make-system (e.g. Automake or custom makefiles), it might be useful for you to use the command:

```
make omkize
```

This will find all `Makefile.omk` files in all subdirectories and copies generic `Makefile` from the root directory to that subdirectories. This way you can easily switch your project to use OMK.

`V` [Variable]

If this variable equals to '`1`', the whole command lines for all executed commands are displayed. When not set or zero, only short messages are printed. Value of '`2`' displays the whole command lines as with '`1`' and in addition directory navigation messages are printed.

2.3 Compiling Programs

To tell OMK to compile a program, you need to set some variables in `Makefile.omk` (usually) in the directory where program sources are located.

In the example bellow program `test` will be compiled from source `test.c`.

```
bin_PROGRAMS = test
test_SOURCES = test.c
```

The variables are:

`bin_PROGRAMS` [Variable]
: Contains a list of names (whitespace separated) of programs to be compiled in this directory.

`test_PROGRAMS` [Variable]
: Almost the same as [bin_PROGRAMS], page 5, but resulting binaries are stored in `bin-tests` directory instead of `bin`. This variable is intended for various test programs not to be mixed with the final product.

`utils_PROGRAMS` [Variable]
: Almost the same as [bin_PROGRAMS], page 5, but resulting binaries are stored in `bin-utils` directory instead of `bin`. This variable is intended for various development utilities not to be mixed with the final product.

`xxx_SOURCES` [Variable]
: For every program name *xxx* in `bin_PROGRAMS`, `test_PROGRAMS` or `utils_PROGRAMS`, this variable contains a list of sources that are needed to compile the program. OMK uses an extension of the filename to determine the compiler to compile this source.

`xxx_LIBS` [Variable]
: This variable contains a list of libraries the program *xxx* will be linked with.

```
test_LIBS = m curses
```

`xxx_CFLAGS` [Variable]
: CFLAGS specific for the compiler program. If this variable is set, its value efectively overrides the value of OMK_CFLAGS variable.

`xxx_CXXFLAGS` [Variable]
: CXXFLAGS specific for the compiler program. If this variable is set, its value efectively overrides the value of OMK_CXXFLAGS variable.

`xxx_CPPFLAGS` [Variable]
: CPPFLAGS specific for the compiler program. If this variable is set, its value efectively overrides the value of OMK_CPPFLAGS variable.

`xxx_GEN_SOURCES` [Variable]
: Program sources generated (by other rules) in the build directory. See the following example.

```
bin_PROGRAMS = p
p_GEN_SOURCES = gen.c

gen.c:
        echo "int main() { return 0; }" > $@
```

`lib_LOADLIBES` [Variable]
: This variable contains a list of libraries which needs to be linked to to all programs or shared libraries in this directory.

`LOADLIBES` [Variable]
: This variable contains a list linker switches to load additional libraries. You usually specify here -L and -l switches.

  Note: The value of this variable is not used used by OMK for any purpose other than linker invocation. Therefore dependency handling of shared libraries does not work if the library is specified in LOADLIBES instead of lib_LOADLIBES.

2.4 Compiling Libraries

With OMK, you can easily create statically or dynamically linked libraries. The way of creating libraries is very similar to how programs are created. See Section 2.3 [Compiling Programs], page 5.

In `Makefile.omk`, you specify several variables, which defines how the libraries should be compiled. In the example below the library '`mylib`' (full filename will be `libmylib.a`) is created from two sources `funca.c` and `funcb.c`. Interface of this library is defined in `myfunc.h`. Therefore, we export this header for use by other programs.

```
lib_LIBRARIES = mylib
mylib_SOURCES = funca.c funcb.c
include_HEADERS = mylib.h
```

Variables for use with libraries are:

`lib_LIBRARIES` [Variable]
: Specifies a list of statically linked libraries to be compiled. OMK automatically prepends `lib` prefix library names.

`shared_LIBRARIES` [Variable]
: Specifies a list of dynamically linked libraries to be compiled.

`xxx_SOURCES` [Variable]
: For every library name *xxx* in `lib_LIBRARIES` or `shared_LIBRARIES`, this variable contains a list of sources that are needed to compile the library. OMK uses an extension of the filename to determine the compiler to compile this source.

2.4.1 Header Files

C and C++ libraries are not very useful without header files. OMK provides several variables that control operations with header files.

During compilation, header files are copied (or linked by symbolic links) from source directories to the `_compiled` tree (see [include-pass], page 3). Libraries and programs are

then compiled against these copies. The following variables control which headers are copied and what is their destination file name.

`include_HEADERS` [Variable]
: Specifies the list of header files to be exported for use by other libraries/programs. The files are exported directly to the `include` directory even if the file is located in a subdirectory (like `sci_regs.h` in the example below)

```
include_HEADERS = regs.h periph/sci_regs.h
```

`nobase_include_HEADERS` [Variable]
: Similar to [include_HEADERS], page 7, but the directory prefix is always kept. To include the file exported by this variable, use `#include <prefix/header.h>`.

`renamed_include_HEADERS` [Variable]
: Exports the header files under different name. The form of the items in this whitespace separated list is: *real name*`->`*new name*.

```
renamed_include_HEADERS = orte_config_omk_win32.h->orte_config.h
```

`LN_HEADERS` [Variable]
: If this variable equals to '`y`', symbolic links to headers in source directories are used in `_compiled` tree instead of copies.

  Normally, the header files are copied into `_compiled` directory to be prepared for transfer into target location afterwards. Copying ensures that resulting libraries are in correspondence with the header files even if the header is changed by a developer but the library is not recompiled.

  On the other side, the copying could make problems during development. Most IDEs, allows you to jump directly to the place, where an error is reported by the compiler. If the error is in a header file, IDE opens you the copy of the header file. If you correct the error there, after the next compilation, your header file will be overwritten by the old version from your source tree.

  This option is not typically used in `Makefile.omk`, but in the top level configuration file `config.omk` or on command line.

2.5 Compiler Flags

OMK follows the same philosophy for flag variables as does Automake. The variables with OMK_ prefix (e.g. OMK_CPPFLAGS) are supposed to be used by the package developer and variable without that prefix (e.g. CPPFLAGS) are reserved for the user. The following

`OMK_CPPFLAGS` [Variable]
: Preprocessor switches.

`OMK_CFLAGS` [Variable]
: C compiler switches.

`INCLUDES` [Variable]
: Directives passed to the C or C++ compiler with additional directories to be searched for header files. In most cases you need to specify an absolute path. To specify a

directory relative to the source directory, you can use the `$(SOURCES_DIR)` variable, which refers to the directory, where `Makefile.omk` is located. This variable applies to all compilations invoked in the current directory.

```
INCLUDES = -I$(SOURCES_DIR)/my_include_dir
```

`DEFS` [Variable]

Directives passed to the C or C++ compiler with preprocessor macro definitions. This variable applies to all compilations invoked in the current directory.

```
DEFS = -DDEBUG=1
```

2.6 Recursing into Subdirectories

OMK is probably most useful in projects consisting of multiple directories. For such projects, it is not easy to write from scratch classic Makefiles that provides all the needed features.

You can instruct OMK to descend to a (sub)directory by setting the `SUBDIRS` variable in `Makefile.omk`.

`SUBDIRS` [Variable]

This variable contains a list of directories, in which compilation must be also invoked. Usually, names of subdirectories are used, but you can use any path specification here.

Compilation is invoked in these directories before it is invoked in the current directory.

See also [AUTOMATIC_SUBDIRS], page 8.

`ALL_OMK_SUBDIRS` [Variable]

This variable is set by OMK and can be used as the value of `SUBDIRS` variable. It contains a list of all direct subdirectories, which contain `Makefile.omk`. This is especially useful if you are combining several projects or components together. In the root directory of your project, you just create symbolic links the components from other projects and all the linked directories automatically appears as the value of this variable.

```
SUBDIRS = $(ALL_OMK_SUBDIRS)
```

`AUTOMATIC_SUBDIRS` [Variable]

If this variable is set to '`y`' and `SUBDIRS` is not assigned in `Makefile.omk`, then `SUBDIRS` is assigned a default value `$(ALL_OMK_SUBDIRS)`.

2.7 Dependency Tracking

OMK automatically tracks dependencies of files in the project. Dependencies of object files are produced with gcc's `-Mx` options. This means that whenever a header file is changed, OMK recompiles only those files, which included that file.

Dependencies are also maintained for libraries and binaries. To find the dependencies, OMK parses linker map files, so a change to some library causes relinking of all programs using that library.

2.8 Configuration and Conditional Compilation

In many projects, it is necessary to configure the compilation process. By this configuring we mean, setting some parameters that influence the output of compilation process. In GNU projects, `configure` script is usually responsible for configuration. User provides some parameters to `configure`, which is run before compilation, and this script does all steps needed to configure the sources and make-system in the desired way.

OMK has its own configuration mechanism, which is described in this section. For future releases, we plan that this mechanism can make use of GNU Autoconf, but currently there is no directly integrated support for it.

There exist three different configuration files `config.omk-default`, `config.target` and `config.omk`. All of these have to be stored in the same directory as `Makefile.rules`. During compilation, these files are included in `Makefile.rules` in this order which means that variables assigned in the former files are overridden by those from later ones. All settings specified here apply to the whole compilation tree. Each file is intended for a different kind of configuration values:

`config.omk-default`
: Stores default configuration of compiled components. This file is automatically generated (see below) and should not be edited by users.

`config.target`
: Stores default configuration for a project or target hardware. This file is intended to be stored in a version control system and should be modified only by the maintainer of the project.

  For cross compiled projects, this file typically contains settings of variables like *CC* and *CFLAGS*.

`config.omk`
: This is a file for end users, where any default settings set in the above files can be overridden. This file should not be stored in version control system. The project should compile without having this file.

Besides variables defined in `config.target`, `Makefile.omk` in any subdirectory can specify some configuration parameters. When `make default-config` is run, all these parameters are found and together with their default values are stored as makefile variables in `config.omk-default`. This file is included during compilation, so if you don't specify other values, these defaults are used. If you are not satisfied with these defaults, you can override the values of parameters either locally for your build in `config.omk` or globally for all people working with the project in `config.target`.

2.8.1 Specifying Configuration Parameters

To specify names and default values of configuration parameters use the `default_CONFIG` variable in `Makefile.omk`.

`default_CONFIG` [Variable]
: This variable contains a list of configuration parameters and their default values. The format of every item in this list is *CONFIG_xxxx=value*. You can name the parameter as you want, but it is good practice to start the name with '`CONFIG_`' prefix.

OMK can automatically generate header files, with C preprocessor macro definitions according to the OMK's configuration parameters. The actual content of generated header files depends on the form of the *value*. The possible forms are:

`'y', 'n' or 'x'`
: This defines boolean parameters. If the value of the parameter is '`y`', the '`#define CONFIG_xxx 1`' is generated, if it is '`n`', no `#define` is generated.

  '`x`' is a special value called *recessive 'n'*. The meaning is that this parameter influences the component in the current directory (i.e. the corresponding `#define` will be included in `LOCAL_CONFIG_H`; see [LOCAL_CONFIG_H], page 11) but the default value is not specified here. If the default value is not specified anywhere, the behavior is the same as if '`n`' is specified.

`'number'`
: Numeric parameters. The define looks like '`#define CONFIG_xxx number`'

`'text'`
: Text without quotes. The define looks like '`#define CONFIG_xxx text`'

`'"text"'`
: Text with quotes. The define looks like '`#define CONFIG_xxx "text"`'

Example of using `default_CONFIG`. `Makefile.omk` reads like:

```
default_CONFIG = CONFIG_DEBUG=y CONFIG_SLOW=n
default_CONFIG += CONFIG_NUM=123 CONFIG_ARCH=arm
default_CONFIG += CONFIG_QUOTES="Text+quotes"

SUBDIRS=subdir
```

and `subdir/Makefile.omk` like:

```
default_CONFIG = CONFIG_SUBDIR=y CONFIG_DEBUG=x
```

After running `make default-config`, the content of `config.omk-default` will be:

```
# Start of OMK config file
# This file should not be altered manually
# Overrides should be stored in file config.omk

# Config for subdir
CONFIG_SUBDIR=y
#CONFIG_DEBUG=x
# Config for
CONFIG_DEBUG=y
CONFIG_SLOW=n
CONFIG_NUM=123
CONFIG_ARCH=arm
CONFIG_QUOTES="Text+quotes"
```

2.8.2 Using Configuration Parameters

Configuration parameters can be used in two ways:

1. as variables in `Makefile.omk` and
2. as C/C++ preprocessor macros in OMK generated header files.

For the first use, your `Makefile.omk` may contain something like:

```
SUBDIRS = arch/$(CONFIG_ARCH)

ifeq ($(CONFIG_DEBUG),y)
DEFS += -DUSE_SIMULATOR
endif
```

For the second use, there are several variables that control the generation of header files with configuration values. These variables are described here:

`LOCAL_CONFIG_H` [Variable]

The value of this variable is the name of a header file, which will contain all configuration parameters declared in the current directory by `default_CONFIG`. This header file is accessible only by files in the current directory and it should be included like `#include "myconfig.h"`.

In `Makefile.omk`, the use of this variable can look like this:

```
LOCAL_CONFIG_H = myconfig.h
```

`config_include_HEADERS` [Variable]

This variable is similar to `LOCAL_CONFIG_H`. One difference is that the generated header file is accessible to all sub-projects in all directories, not only to the files in the same directory (the header is stored in `_compiled` tree). The second difference is that you have to specify, which configuration parameters you want to appear in the header file.

`xxx_DEFINES` [Variable]

This variable determines the configuration parameters that should be stored in a header file specified by `config_include_HEADERS`. The *xxx* in the name of this variable needs to be the same as the base name (without extension) of the header file.

Example of using `config_include_HEADERS`:

```
default_CONFIG = CONFIG_LINCAN=y CONFIG_LINCANRTL=n CONFIG_LINCANVME=n
config_include_HEADERS = global.h
global_DEFINES = CONFIG_OC_LINCAN CONFIG_OC_LINCANRTL
```

Here, we include only two out of the three configuration parameters defined in the current `Makefile.omk`. It is also possible to include configuration parameters defined in a different directory.

2.8.3 Common Variables

It is common practice to use `config.target` or `config.omk` to store project-wide settings. Here is the list of variables, which are commonly set here (but they can also be set elsewhere, e.g. in `Makefile.omk`).

You can easily "reconfigure" your project by changing the `config.omk` file. It is useful to have several configurations stored in different files and let `config.omk` be a symbolic link to the desired configuration.

`CC` The name of C compiler.

`CFLAGS` Command line options for C compiler.

`CXX` The name of C++ compiler.

`CPPFLAGS` Additional parameters (besides `CFLAGS`) to by passed to C++ compiler.

2.9 Advanced OMK Features

In this section we list several OMK features, which are more complicated or rarely used so they were omitted in previous sections.

2.9.1 Organization of the Source Tree

- The `_compiled` directory can be shared between multiple projects (by using symbolic links).
- If you work on a bigger project, you usually don't need to rebuild the whole project and call `make` only in a subdirectory. Sometimes, it might be useful to rebuild the whole project. You can either change working directory to the root of your project and call `make` there or, as a shortcut, you can use `W` variable (see [W], page 12) to compile everything directly from a subdirectory.

```
make W=1
```

- Searching for `Makefile.rules` works such way, that if you get into sources directory over symbolic links, OMK is able to unwind your steps back. This implies you can make links to component directories on read-only media, copy `Makefile.rules`, `Makefile` and top-level `Makefile.omk`, adjust `Makefile.omk` to contain only required components and then call `make` in the top directory or even in read-only directories after changing working directory from your tree to the readonly media.

`W` [Variable]

If this variable equals to '`1`', the **whole** project is (re)compiled, even if `make` is called from a subdirectory.

2.9.2 Additional Variables

`USE_LEAF_MAKEFILES` [Variable]

If this variable equals to '`n`' (default is unset), then OMK uses the leaf `Makefile` only when it is invoked by simple `make` command. Later, during recursive directory descent leaf `Makefile` is not used and `Makefile.rules` is included directly.

This feature is useful if you are integrating some non-OMK project into your project. You only add `Makefile.omk` files to the non-OMK project and don't need to modify project's original Makefiles.

This variable can be set either globally in a `config.*` file or locally in some `Makefile.omk`. In the latter case, it influences only subdirectories of the directory containing `Makefile.omk`.

`SOURCES_DIR` [Variable]

This variable is set internally by OMK and its value is the absolute path to the directory with compiled sources. It can be used if you need to refer to sources files in some custom constructs in `Makefile.omk`.

```
include_HEADERS = $(notdir $(wildcard $(SOURCES_DIR)/*.h))
```

`srcdir` [Variable]
The same as [SOURCES_DIR], page 12. Provided for Automake compatibility.

`MAKERULES_DIR` [Variable]
This variable is set internally by OMK and its value is the absolute path to the directory containing `Makefile.rules` currently used during compilation.

`OMK_RULES_TYPE` [Variable]
Identification the type of `Makefile.rules` used for compilation. Values are like '`linux`', '`rtems`', '`sysless`', ... This variable is automatically generated during creation of `Makefile.rules` and can be used in configuration files (see Section 2.8 [Configuration and Conditional Compilation], page 9) or in `Makefile.omk` to tweak compilation for specific targets.

2.9.3 Adding Hooks to Passes

Sometimes it is necessary to run some special commands as a part of compilation. Typical example might be a tool which generates source files on the fly. OMK supports calling additional commands during compilation by so called *pass hooks*. A pass hook is an ordinary make target which is invoked as part of compilation during a particular pass (see [passes], page 3). Pass hooks can be defined by assigning their names to `xxx_HOOKS` variable.

`xxx_HOOKS` [Variable]
Specifies one or more hooks (make targets) which are invoked during pass *xxx*. The working directory of commands or this target is under the `_build` tree.

In the example bellow header file `generated_header.h` is created during '`include-pass`' by `convert_data` program. The program takes `data_file.txt` in the source directory as the input and creates the header file in the in the correct directory under the `_build` tree.

```
include-pass_HOOKS = generated_header.h

generated_header.h: $(SOURCES_DIR)/data_file.txt
            convert_data < $^ > $@
```

2.9.4 Integration of Wvtest Framework

OMK has integrated support for Wvtest unit testing framework. It is a very minimalistic framework whose integration with OMK does not impose almost any particular policy of writing the tests. Wvtest tests are specified by the following two variables:

`wvtest_PROGRAMS` [Variable]
This variable has the same meaning as [test_PROGRAMS], page 5 with two exceptions: (1) the program is automatically linked with the library specified by `WVTEST_LIBRARY` variable and (2) the program is automatically executed by `make wvtest` (see below).

`wvtest_SCRIPTS` [Variable]
Defines the name of a script (e.g. shell script) which is executed by `make wvtest`. Write the scripts so, that they can be run from arbitrary directory, i.e. in the case of shell scripts ensure that the `wvtest.sh` library is sourced like this:

```
. $(dirname $0)/wvtest.sh
```

`WVTEST_LIBRARY` [Variable]
: Specifies the name of the library that is automatically linked with `wvtest_PROGRAMS`. The default is `wvtest`.

There is also an OMK pass called `wvtest-pass` which consecutively runs all `wvtest_PROGRAMS` and `wvtest_SCRIPTS` in the traversed subdirectories of the current directory. Every program or script is executed in a temporary directory under `_build` directory with `PATH` variable modified to include `_compiled/bin` as the first component and `LD_LIBRARY_PATH` modified to include `_compiled/lib` as the first component. This allows the tests to run the `bin_PROGRAMS` without explicitly specifying their full path and to use shared libraries without the path as well.

When make is invoked as `make wvtest` it runs `make wvtest-pass` under the control of `wvtestrun` script, which must be present in the same directory as `Makefile.rules`. This script suppresses the normal output of passed tests and prints only their summary. For failed tests, the full output is shown. Additionally, when the output is written to a terminal, the status of each test is displayed in color for easy inspection.

2.10 Properties of Specific Makefile.rules

In previous sections, general properties of `Makefile.rules` were documented. This section contains documentation to features found only in some particular `Makefile.rules`.

2.10.1 Linux

This `Makefile.rules` is used not only for Linux as the name suggests, but also for other Unices and even for Windows.

`BUILD_OS` [Variable]
: The name of the operating system (OS) where make was invoked.

`TARGET_OS` [Variable]
: Should specify the name of OS where the resulting binary should be used. If not specified manually, it equals to BUILD_OS.

`QT_SUBDIRS` [Variable]
: Lists subdirectories with QT project (.pro) file. OMK will generate there `Makefile` by calling `qmake` with correct parameters to interface QT application to the rest of the compilation tree. Then `make` is called there to compile QT application. Variable '`QTDIR`' must be set to the directory with QT installation (e.g. /usr/share/qt4 on Debian).

`bin_SCRIPTS` [Variable]
: Lists the names of the files (persumably scripts) to be copied to `_compiled/bin`.

2.10.2 System-Less

This `Makefile.rules` is designed for compilation of code for (small) micro-controllers without operating systems. See `http://rtime.felk.cvut.cz/hw/index.php/System-Less_Framework` for more information about our framework, which uses this rules.

2.10.3 RTEMS

TODO

2.11 Running OMK under Windows OS

It is possible to use OMK under Windows OS with MinGW (see http://www.mingw.org/). Unfortunately, the compilation speed is much lower than on UNIX systems.

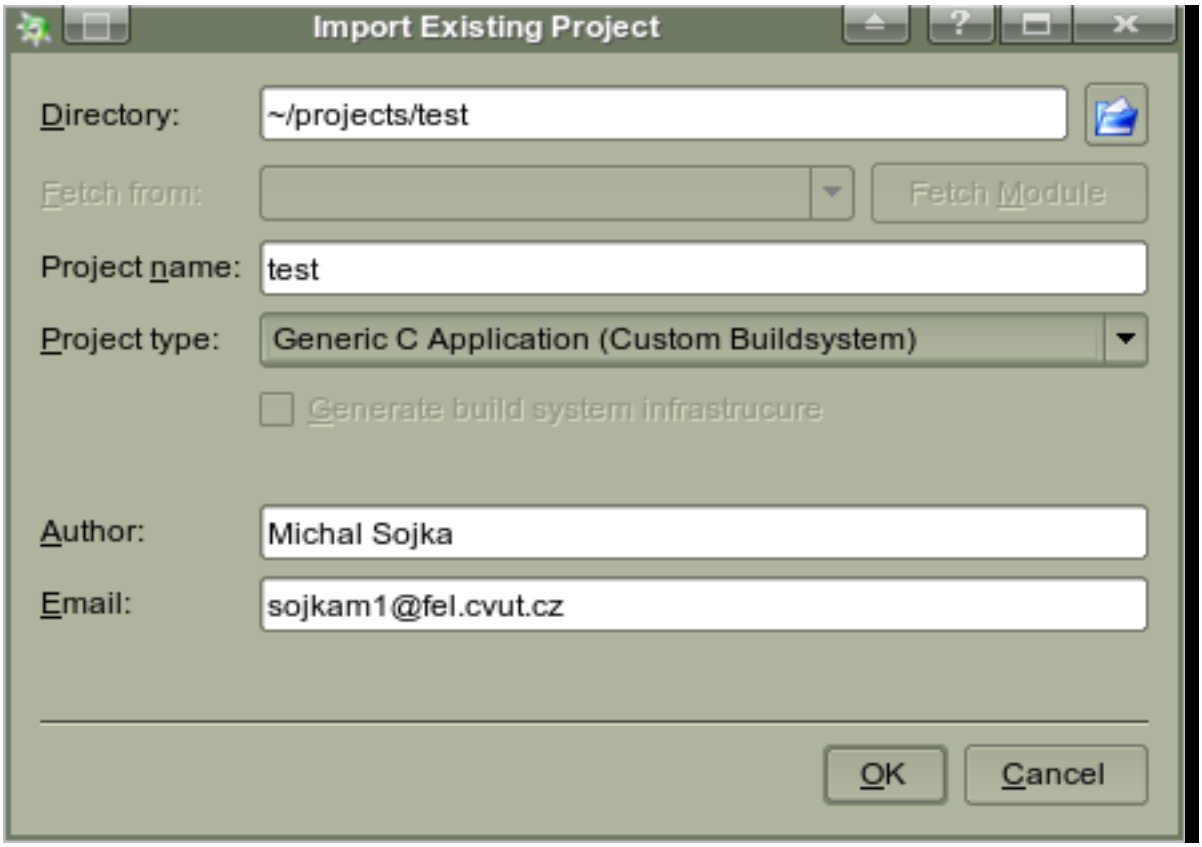
TODO: Is it necessary to install anything special?

2.12 Interfacing OMK to popular IDEs

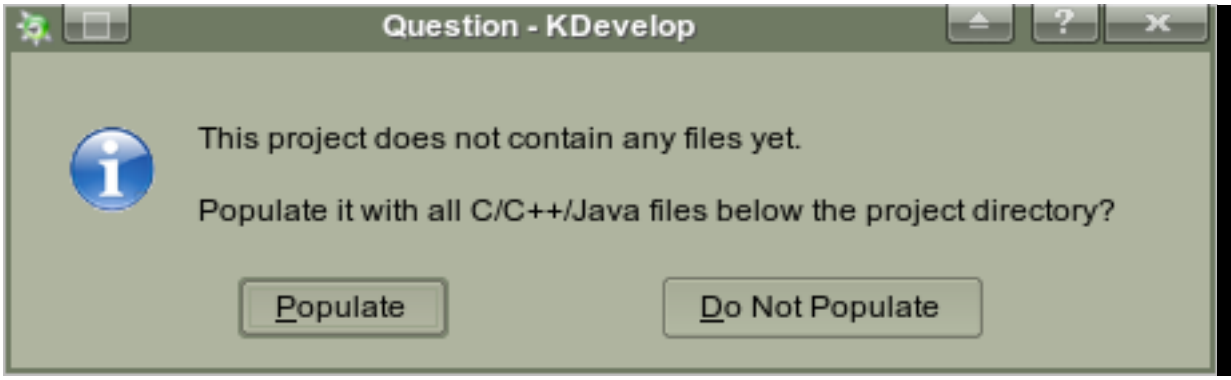
2.12.1 KDevelop

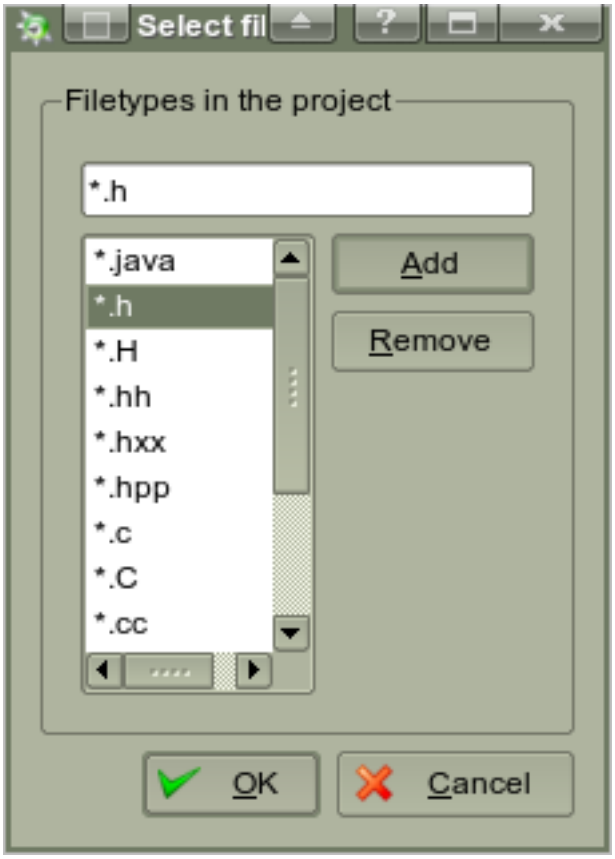
KDevelop has support for custom build systems. To use KDevelop to develop projects using OMK follow these steps. These steps are valid for version 3.5.0 of KDevelop, but for previous versions it doesn't differ much.

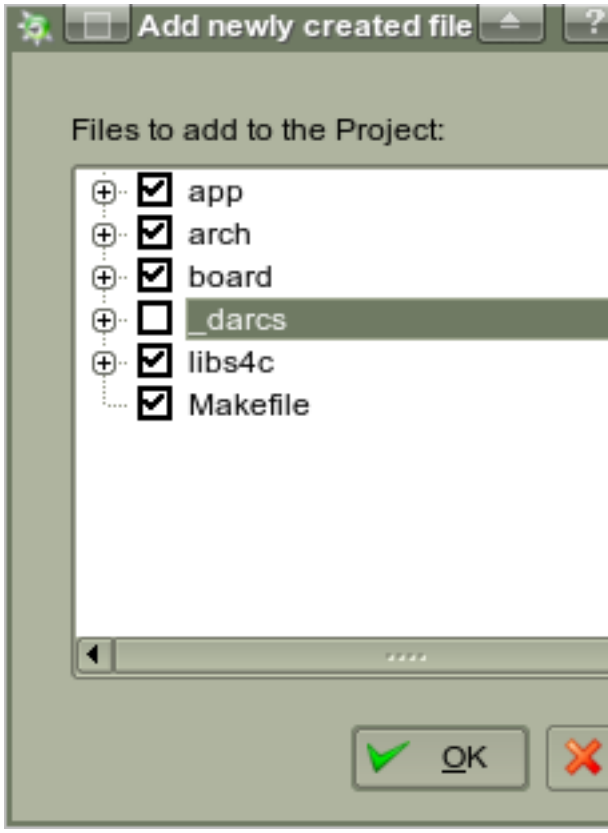
1. Import project to KDevelop (from menu choose *Project—Import existing project*). Select the type of project to *Generic C Application (Custom Buildsystem)*.



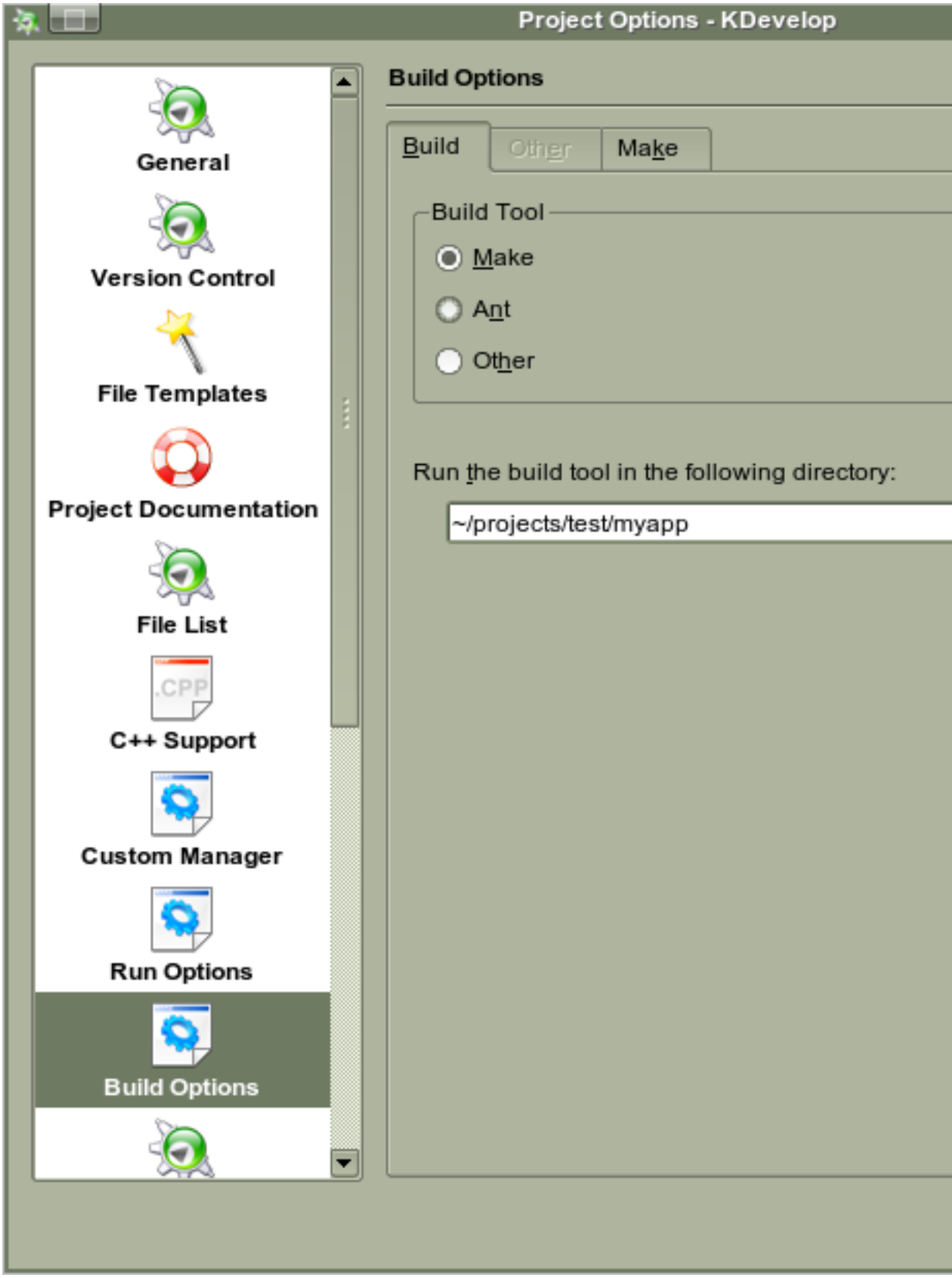
2. Then answer to following dialogs as you want.





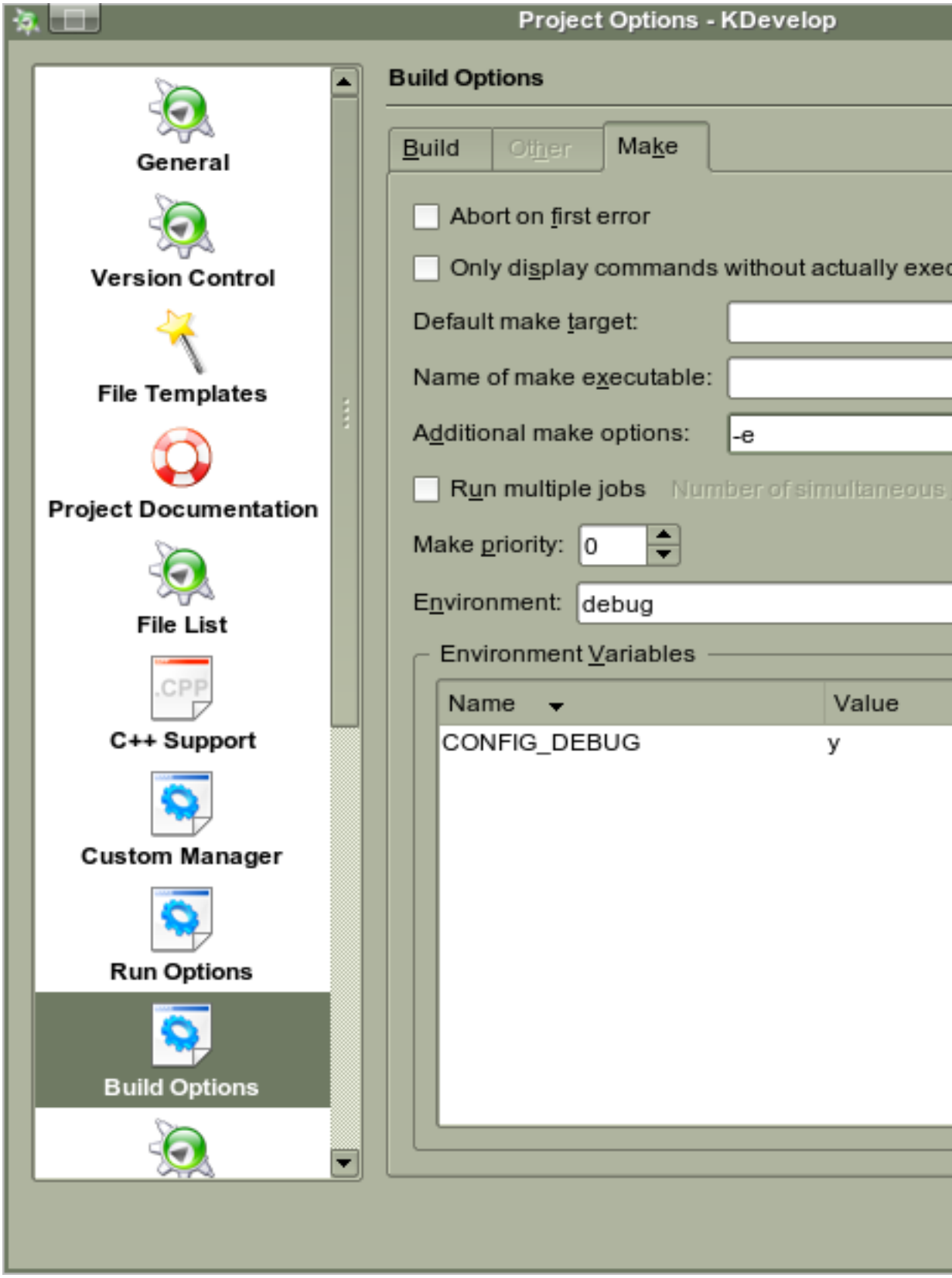


3. If you are working only on some small part of the bigger project, you usually don't want to recompile the whole project every time. In *Project—Project Options*, you can specify the subdirectory where to run `make`.

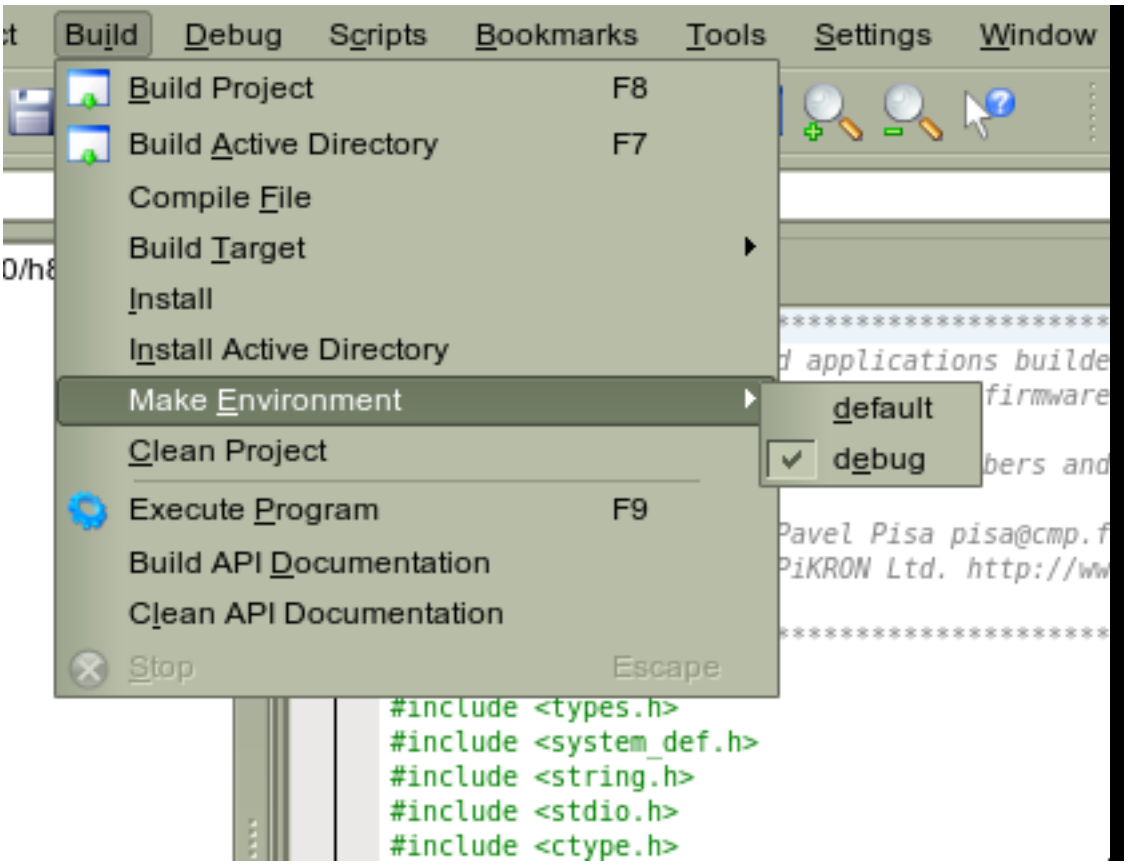


4. If you want to switch between several configurations easily (see also Section 2.8 [Con-

figuration and Conditional Compilation], page 9), in the same dialog you can add `-e` to make options. This makes environment variables have higher precedence than those in `config.omk-default`. Then, you can define several environments with different `CONFIG_`*`xxx`* variables and their values.



5. You can easily switch the configurations from *Build—Make Environment*.



2.12.2 Eclipse/CDT

TODO

2.12.3 Emacs, VIM, etc.

Since OMK compilation is started by executing `make` command, many common editors can work easily with OMK.

Under Emacs, you can use `compile` or `recompile` commands as you are used to do.

2.13 Troubleshooting & Knows Bugs

- If you rename some file or directory and then you can't compile your project, call `make clean` in the directory with errors. The reason for this behavior is that OMK remembers dependencies of every file. After renaming something, the original name is still stored in dependencies, but make doesn't know how to create this non-existent source.
- Sometimes, you may want to compile one file the same way as OMK does it, but run the compilation manually from command line. For example, you want to debug some preprocessor macros and you only want to produce preprocessed source instead of an object file.

  To compile something manually, you can run OMK by `make V=2`. This will print all commands executed together with directory navigation messages. Find the command you want to execute manually in the output. To run it, you need to change the working

directory to the correct one in the `_build` tree. The correct directory can be found in make output on the line '`Entering directory`' preceding the desired command.

- Currently, C++ sources are supposed to have `.cc` or `.cxx` extensions. The `.cpp` extension is not supported (yet).
- Removing of library source file does not cause the library to be rebuild. You need to manually delete the library or run `make distclean` before you run `make`.

3 Original README

Since this manual still doesn't cover all aspects of OMK, we include here a `README.rules` file, which was written for the first version of OMK.

Important notice: This make system uses features found in recent versions of GNU Make program. If you encounter problems with package building, check, that you use correct version of Make program. The Make older than version 3.80, could not be used. Even Make version 3.80 has annoying bug which causes building fail with misleading message "virtual memory exhausted". Please, upgrade at least to version 3.81 of GNU Make.

There is list of features which we want to solve with our make system:

- Central `Makefile.rules` for most of components of a bigger project.

  FIXME (our CAN framework includes more libraries common with our other projects, we need to separate some utility libraries etc.)
- The rules in more spread Makefiles are way to the hell (update for different kernel, RT-Linux etc would be nightmare in other case).
- Make system should allow to freely move cross-dependant components in directory structure without need to update users of moved component (I hate something like `-I../../sched/rtlshwq/include` in CAN makefiles for example. If a component is renamed or version is added to then name, all Makefiles in CAN will require update).
- Make system should be able to compile mutually cross-dependant libraries and should ensure, that change in one component sources or headers would result in relink or rebuild in components linked against that library or including modified header file.
- Make system has to enable compilation out of OCERA full source tree (we would lost many users of particular components in other case).
- Compile should be able to do all above work without need to install any files before successful finish of build.
- Because we use some libraries for RT-Linux build and user-space build, we need to solve how to compile from same sources to both targets.
- The build system should allow to call make for particular source subdirectory. Time of recursive make through all subdirectories is unacceptable.
- Make system should enable to build out of sources tree (else clean or working with CVS sandbox gets fussy and simultaneous multiple targets gets problematic).
- It would be good, if there is a possibility to call make from read-only media sources.
- Make system should store results of build in some separate directory structure to simple install and testing.
- Makefiles in sources directories should be simple.

There is probably only one alternative fully supporting above requirements and it is GNU Autoheader...Automake...Autoconf... system. But it is complicated and requires big amount of support files. It would be acceptable if it could be easily used for OCERA framework. But there are important show stoppers for that system:

- It would require deep revision of all OCERA CVS contents and agreement on this would be problematic

- This system is not well prepared for dual compilation for Linux and RT-Linux subtargets. It would mean many changes in default autoconf setup to support this. Probably simplest way would be to rebuild GCC tool chain for something like i586-elf-rtlinux. This would require even more space for OCERA development.

The problem calls for some solution, which would have minimal impact on other components and would be elegant and would be maintainable and small, because our main goal is components development and not make systems development.

There is result of our trial. It is OMK make system. The `Makefile` and `Makefile.omk` files should be in all source directories. Common `Makefile.rules` file is required in the toplevel sources directory. Alternatively this file could be moved to link tree pointing into readonly media or can be anywhere else if `MAKERULES_DIR` and `SOURCES_DIR` are specified.

Syntax of Makefile.omk files is for usual cases compatible to Automake's Makefile.am descriptions. There are specific targets for RT-Linux and Linux kernel related stuff

Makefile.omk user defined variables

`SUBDIRS` list of subdirectories intended for make from actual directory

`lib_LIBRARIES`
: list of the user-space libraries

`shared_LIBRARIES`
: list of the user-space shared libraries

`kernel_LIBRARIES`
: list of the kernel-space libraries

`rtlinux_LIBRARIES`
: list of the RT-Linux kernel-space libraries

`include_HEADERS`
: list of the user-space header files

`nobase_include_HEADERS`
: headers copied even with directory part

`kernel_HEADERS`
: list of the kernel-space header files

`rtlinux_HEADERS`
: list of the RT-Linux kernel-space header files

`bin_PROGRAMS`
: list of the require binary programs

`utils_PROGRAMS`
: list of the development utility programs

`kernel_MODULES`
: list of the kernel side modules/applications

`rtlinux_MODULES`
: list of RT-Linux the kernel side modules/applications

`xxx_SOURCES`
: list of specific target sources

`INCLUDES` additional include directories and defines for user-space

`kernel_INCLUDES`
: additional include directories and defines for kernel-space

`rtlinux_INCLUDES`
: additional include directories and defines for RT-Linux

`default_CONFIG`
: list of default config assignments CONFIG_XXX=y/n ...

The Makefile is same for all sources directories and is only 14 lines long. It is there only for convenience reasons to enable call "make" from local directory. It contains code which locates `Makefile.rules` in actual or any parent directory. With standard BASH environment it works such way, that if you get into sources directory over symbolic links, it is able to unwind yours steps back => you can make links to readonly media component directories, copy `Makefile.rules`, Makefile and toplevel Makefile.omk, adjust Makefile.omk to contain only required components and then call make in top or even directories after crossing from your tree to readonly media.

The system compiles all files out of source directories. The actual version of system is adapted even for OCERA tree mode if `OCERA_DIR` variable is defined in `Makefile.rules`

There are next predefined directory name components, which can be adapted if required

`BUILD_DIR_NAME = _build`
: prefix of directory, where temporary build files are stored

`COMPILED_DIR_NAME = _compiled`
: prefix of directory, where final compilation results are stored

`GROUP_DIR_NAME = yyy`
: this is used for separation of build sub-trees in OCERA environment where more `Makefile.rules` is spread in the tree

Next directories are used:

`KERN_BUILD_DIR := $(MAKERULES_DIR)/$(BUILD_DIR_NAME)/kern`
: directory to store intermediate files for kernel-space targets

`USER_BUILD_DIR := $(MAKERULES_DIR)/$(BUILD_DIR_NAME)/user`
: directory to store intermediate files for user-space targets

`USER_INCLUDE_DIR := $(MAKERULES_DIR)/$(COMPILED_DIR_NAME)/include`
: directory to store exported include files which should be installed later on user-space include path

`USER_LIB_DIR := $(MAKERULES_DIR)/$(COMPILED_DIR_NAME)/lib`
: same for user-pace libraries

`USER_UTILS_DIR := $(MAKERULES_DIR)/$(COMPILED_DIR_NAME)/bin-utils`
: utilities for testing, which would not probably be installed

`USER_BIN_DIR := $(MAKERULES_DIR)/$(COMPILED_DIR_NAME)/bin`
: binaries, which should go into directory on standard system PATH (/usr/local/bin, /usr/bin or $(prefix)/bin)

`KERN_INCLUDE_DIR := $(MAKERULES_DIR)/$(COMPILED_DIR_NAME)/include-kern`
: directory to store exported include files which should be installed later on kernel-space include path

`KERN_LIB_DIR := $(MAKERULES_DIR)/$(COMPILED_DIR_NAME)/lib-kern`
: same for kernel-pace libraries

`KERN_MODULES_DIR := $(MAKERULES_DIR)/$(COMPILED_DIR_NAME)/modules`
: builded modules for Linux kernel or RT-Linux system

There is more recursive passes through directories to enable mutual dependant libraries and binaries to compile. Next passes are defined

'`default-config`'
: generates `config.omk-default` or xxx-default (FIXME) configuration file

'`check-dir`'
: checks and creates required build directories

'`include-pass`'
: copies header files to `USER_INCLUDE_DIR` and `KERN_INCLUDE_DIR`

'`library-pass`'
: builds objects in USER_BUILD_DIR/*relative path* and creates libraries in USER_LIB_DIR

'`binary-pass and utils-pass`'
: links respective binaries in USER_{BIN,UTILS}_DIR directory. If some object file is missing it compiles it in USER_BUILD_DIR/*relative path*

'`kernel-lib-pass`'
: builds libraries for kernel space targets

'`kernel-pass`'
: builds kernel modules

The amount of passes is relatively high and consumes some time. But only other way to support all required features is to assemble one big toplevel Makefile, which would contain all components and targets cross-dependencies.

Drawbacks of designed make system

- the system is not as fast as we would like
- it lacks Autoconf and configure extensive support for many systems from UNIX to DOS and WINDOWS
- it does not contain support for checking existence of target libraries and functionalities as GNU Autoconf
- it is heavily dependant on GNU MAKE program. But it would not be big problem, because even many commercial applications distribute GNU MAKE with them to be able to work in non-friendly systems

- the key drawback is dependence on recent MAKE version 3.80 and better and even version 3.80 of MAKE has important bug, which has been corrected in newer sources (FIXME)

The last point is critical. I have not noticed it first, because I use Slackware-9.2 and it contains latest released version of MAKE (version 3.80). The problem appears when I have tried to build bigger libraries. There is bug in version 3.80, which results in misleading error "Virtual memory exhausted". It is known bug with ID 1517

```
* long prerequisite inside eval(call()) => vm exhausted, Paul D. Smith
```

I have optimized some rules to not push memory to the edge, but there could be still issues with 3.80 version.

I have downloaded latest MAKE CVS sources. The compilation required separate lookup and download for .po files and full Autoheader... cycle. I have put together package similar to release. Only ./configure –prefix=... and make is required. CVS sources contains version 3.81beta1. You can download prepared sources archive from http://paulandlesley.org/make/make-3.81beta1.tar.bz2 Or you can get our local copy from http://cmp.felk.cvut.cz/~pisa/can/make-3.81beta1.tar.gz

The archive contains even "make" binary build by me, which should work on other Linux distributions as well. Older version of MAKE (3.79.x released about year 2000) found on Mandrake and RedHat are not sufficient and do not support eval feature. I do not expect, that Debian would be more up-to-date or contain fixes to MAKE vm exhausted bug.

The local CTU archive with our CAN components prepared for inclusion into OCERA SF CVS could be found in my "can" directory

http://cmp.felk.cvut.cz/~pisa/can/ocera-can-031212.tar.gz

The code should build for user-space with new make on most of Linux distros when make is updated.

If you want to test compile for RT-Linux targets, line

```
#RTL_DIR := /home/cvs/ocera/ocera-build/kernel/rtlinux
```

in `Makefile.rules` has to be activated and updated to point RT-Linux directory containing "rtl.mk". There is only one library ("ulutrtl") and test utility compiled for RT-Linux (`can/utils/ulut/ul_rtlchk.c`).

The next line, if enabled, controls compilation in OCERA project tree

```
#OCERA_DIR := $(shell ( cd -L $(MAKERULES_DIR)/../../.. ; pwd -L ) )
```

The LinCAN driver has been updated to compile out of source directories.

Please, check, if you could compile CAN package and help us with integration into OCERA SF CVS. Send your comments and objections.

The OMK system has been adapted to support actual OCERA configuration process. I am not happy with ocera.mk mix of defines and poor two or three rules, but OMK is able to overcome that.

The OMK system has integrated rules (default-config) to build default configuration file. The file is named `config.omk-default` for the stand-alone compilation. The name corresponds to OCERA config + "-default" if OCERA_DIR is defined. This file contains statements from all `default_CONFIG` lines in all `Makefile.omk`. The file should be used for building of own `config.omk` file, or as list for all options if Kconfig is used.

4 Development

This section is far from complete. Its purpose is to document internals of `Makefile.rules` as well as other things needed only by people who hack OMK itself.

4.1 Passes

A pass is created by instantiation of `omk_pass_template` with *pass-name* as one of arguments. This defines several targets which are described here:

pass-name
: Target used to invoke the individual pass either from command line or from inside of `Makefile.rules`.

pass-name`-submakes`
: Invoked recursively from *pass-name*. The reason for this is the fact that

pass-name`-this-dir`
: This target calls make recursively once again with *pass-name*-local target, which does the real-work. Make's working directory is set to the corresponding directory in `_build` tree and the -local

pass-name-dirname`-subdir`
: This target is responsible for recursive invocation of `make` in subdirectories specified in [SUBDIRS], page 8 variable.

Variable Index