Introduction

This document describes the MutekH build system invocation and base features and is intended for beginners and application developers.

The build system takes care of features dependencies, file modifications, ... but is still simple to use for the beginner. MutekH source code is highly configurable, this implies the application developer has to provide a **build configuration file** along with the application source code.

Available features, tweakable values and associated constraints are represented by **configuration tokens** and described by kernel developers through **configuration description files** (.config) in the source tree. The configuration description syntax is detailed on the BuildSystemDev page which cover more advanced topics.

Reading the BuildingExamples page may be of some interest too.

Invocation

Building

When building MutekH, several options may be used to change the behavior of the build system. These options are given through variables when calling `make`, in the form:

```
$ make VAR1=value1 VAR2=value2
```

Real build process invocation just need to specify build configuration to use:

When using a flat build configuration file:

```
$ make CONF=examples/hello/config_soclib_mipsel
```

When using a sectioned build configuration files, you need to specify which sections must be considered for the build:

```
$ make CONF=examples/hello/config BUILD=soclib-mips32el
$ make CONF=examples/hello/config BUILD=soclib-arm:debug
```

Depending on the target architecture, the binary output file may be an ELF (.out), a plain binary file (.bin), an intel-hex file (.hex) or an object file (.o).

Configuration display

This section explains how to display information about MutekH configuration being used.

You can display a list of relevant tokens with their value for a given configuration:

```
$ make CONF=path/to/config_file listconfig
```

You can display a list of **all** tokens with their value for a given configuration:

```
$ make CONF=path/to/config_file listallconfig
```

To display help about a specific token:

```
$ make CONF=path/to/config_file showconfig TOKEN=CONFIG_PTHREAD
```

Main variables

The following option is mandatory:

`CONF=`
: An absolute path to the build configuration file.

The following option may be required depending on build configuration file:

`BUILD=`
: A colon separated list of build section names to consider in the build configuration file.

The following options may be useful:

`VERBOSE=1`
: Prints all the commands executed

The following options are useful when building out of the source tree:

`MUTEK_SRC_DIR`
: An absolute path to the MutekH source tree. This defaults to `.`

`BUILD_DIR`
: An absolute path to the directory containing the objects and results, this defaults to `.`

`CONF_DIR`
: An absolute path to the directory containing the `.config.*` files, this defaults to `$(BUILD_DIR)`

Make targets

The following targets are available

`kernel`
: This is the default target. It builds the kernel for the specified configuration file.

`kernel-het`
: This target builds multiple kernels for heterogeneous multiprocessors platforms.

`clean`
: This target cleans all the compilation results

The following targets are for informational purposes

`showpaths`
: This prints the modules that will be built, their paths, ?

`cflags`
: This prints the flags used for compilation

The following targets are available to get help about configuration.

`listconfig`
: Prints the current configuration as expanded by MutekH build system. It also prints available --- but currently undefined --- configuration tokens.

`listallconfig`

Prints all the configuration tokens, even the ones incompatible with the current configuration.

`showconfig`
: This prints detailed information about a given configuration token. Token must be specified with `TOKEN=` variable argument.

See usage below.

Build configuration files

Content

MutekH build configuration files contain token values pairs defining the kernel we are currently building. They must contain:

- the license for the application, enforcing license compatibility between some kernel parts and your code,
- the target architecture
- the libraries used, and their configurations
- the used drivers
- some global compilation switches (optimization, debugging, ?)
- ...

Basic syntax

Syntax is `token` **space** `value`. Tokens begin with `CONFIG_`. Value may be omitted thus defaults to `defined`. e.g.

```
CONFIG_LICENSE_APP_LGPL

# Platform type
CONFIG_ARCH_EMU

# Processor type
CONFIG_CPU_X86_EMU

# Mutek features
CONFIG_PTHREAD

# Device drivers
CONFIG_DRIVER_CHAR_EMUTTY

# Code compilation options
CONFIG_COMPILE_DEBUG undefined

...
```

Most common values are `defined` and `undefined` to enable and disables features, but some tokens may need numerical or string values.

The easiest way to write a configuration file is to rely on and include common sectioned configuration files and just write the minimal application related configuration yourself. See below.

Have a look to hg/examples/hello for examples of working build configuration files.

The MutekH API reference manual describes all available configuration tokens.

Module declaration

MutekH has a modular and component-based architecture.

A build configuration file may declare a new module for the application. Modules can be located anywhere outside of the main source tree. We must tell the build system the directory where the configuration lies. The path to the module directory is usually the same as its configuration file and the `CONFIGPATH` special variable is well suited:

```
# New source code module to be compiled
# %append MODULES name:module_dir
  %append MODULES hello:$(CONFIGPATH)
```

Output name

the MutekH build system takes care of building in directory named after application name and build target. This determine the application output file name too. You may want to set your application output name in build configuration file:

```
%set OUTPUT_NAME hello
```

Advanced syntax

Basic configuration is really simple. Complex applications or multiple target architectures require maintaining multiple configuration files which can be difficult and annoying. The directives presented here are used to make things easier. They are mainly used in common build configuration files found in hg/examples/build.

Sectioning directives are useful to consider a set of configuration definitions depending on the `BUILD` variable of `make` invocation:

`%section pattern [pattern ...]`
: Start a section which will be conditionaly considered depending on the `BUILD` variable. `pattern` is a pattern matching expression which may contain text, hypens and wildcards (e.i. `text-text-*`). Wildcards match non-empty and non-hypens text. A new `%section token automatically terminate previous section.`

`%common`
: Revert to unconditional common file part, default at beginning of a file.

`%else`
: Change current conditional state.

`%subsection [pattern ...]`
: Begin a nested section. Multiple levels of subsections can be used. Subsections thus defined must be end by `%end`.

`%end`
: End a subsection started with `%subsection`.

Section types directives can be used to enforce use of sections:

`%types type [type ...]`
: Specify that the current section exhibits the given types. No more than one section can be in use with the same type.

`%requiretypes type [type ...]`
: All specified types must have been defined. May be used in sections or common part.

Build configuration files may contain variables:

`%set variable content`
: Set a variable which can be expanded using `$(variable)` syntax. Environment is initially imported as variables. Moreover `$(CONFIGPATH)` and `$(CONFIGSECTION)` are predefined special variables.

`%append variable content`
: Appends content to a variable.

Build configuration files may include other files:

`%include filename`
: Include a configuration file, the new file always begin in `%common` state.

Build configuration files may report things to the user:

`%notice text`
: Produce a notice message.

`%warning text`
: Produce a warning message.

`%die text`
: Produce an error message.

`%error text`
: Produce an error message with file location information.

The `default` section name is in use when no section name is passed through the `BUILD` variable.

Some build configuration files are available in examples/common and can be included to target common platforms without having to deals with all configuration token. This helps keeping application configuration file short. Configuration tokens can be redefined multiple times, allowing to override values set in included files.