Introduction

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

The build system is responsible for building MutekH with requested features configuration. 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, options and associated constraints are represented by **configuration tokens** which are declared by kernel developers in **configuration description files** (.config) in the kernel source tree. These files define set the set of configuration tokens which can be assigned in a **build configuration file**. The configuration description syntax is detailed on the BuildSystemDev page which cover more advanced topics related to kernel developpment.

This page focus on the build system invocation and the format of the build configuration file. Reading the BuildingExamples page may be of some interest too.

Invocation

Building

Invocation of the build system is performed by running GNU make at the root of the MutekH source tree.

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
```

Build process invocation actually require the user to specify a build configuration file to use.

Using a flat build configuration file which contains the whole kernel configuration, including supported hardware and device drivers definitions, is the most simple way to invoke the build system:

```
$ make CONF=path/to/config.build
```

This however requires the user to write a long build configuration file which is target specific. Relying on a sectioned build configuration files (see below) make this file short and easier to write. We then need to specify which sections of the file must be considered for the build:

```
$ make CONF=path/to/config.build BUILD=ibmpc-x86
$ make CONF=path/to/config.build BUILD=gaisler-leon3
$ make CONF=path/to/config.build BUILD=soclib-mips32el:pf-tutorial
```

Configuration display

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

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

```
$ make CONF=path/to/config.build BUILD=... listconfig
```

You can display a detailed description of initializations which take place during KernelStartUp; this also depend on the build configuration in use:

```
$ make CONF=path/to/config.build BUILD=... listinit
```

More targets are available, see below.

Build log

When the compilation process ran successfully, a build log is written to a `.log` file named after the generated kernel file. This file contains various information about the configuration used for the build which can be useful to developpers.

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

`TARGET_EXT=`
: Generate a kernel image in various formats: `out` (elf), `o` (object), `bin` (flat binary), `hex` or `srec`.

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_PATH`
: An absolute paths to extra directories containing external source modules.

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 along with associated paths.

`cflags`
: This prints the flags used for compilation.

The following targets are available to get help about configuration.

`listconfig`
: Prints values of configuration tokens which would be used to build MutekH according to current build configuration.

`listallconfig`
: Prints all the configuration tokens, including internal tokens and tokens with undefined parents.

`listflatconfig`
: Prints the content of a minimal flat configuration file which could be used to define current configuration.

`listinit`
: Prints initialization tokens and initilization function calls which will take place during KernelStartUp according to current build configuration.

See usage below.

Build configuration files

Content

The MutekH build configuration file defines token values describing feature included in the kernel which is being built. It must contain:

- the license for the application, enforcing license compatibility between some kernel parts and your code,
- the target architecture
- libraries to build, and related options
- device 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 take numerical or string values.

The easiest way to describe your configuration is to rely on sectioned **build configuration files** which are provided in the kernel source tree. These files are split in multiple sections which are conditionnaly parsed depending on the value of the `BUILD` variable passed to the build system.

They provide default configurations for supported processors and platforms so that you just have to write a minimal file which take care of enabling modules and features which are needed by your application. This way, all configuration token related to target hardware will get their value from those files. This do not prevent the user to override a token value after inclusion of one of those files. The user can also define its own sections on its build configuration, see syntax at the end of this page.

```
...

CONFIG_LICENSE_APP_LGPL

CONFIG_PTHREAD

%include arch/arch.build

CONFIG_CPU_MAXCOUNT 4
```

Have a look to hg/examples/hello for examples of real 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 must 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 file is located. 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

As explained previoudly, basic build configuration format is simple. Complex applications or support for multiple target architectures require maintaining multiple configuration files which can be difficult and annoying. The directives presented here can be used to define conditional sections in build configuration files.

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 provided in the kernel source tree and can be included to target hardware platforms without having to deals with all related configuration token. Configuration tokens can be assigned multiple times, this allows overriding values previously assigned in included files.