This document describes the MutekH build system.

Overview of the build process

The build system use one or more used defined **build configuration files** to compiles the desired kernel together with the application.

The build system takes care of features dependencies, file modifications, ... but is still simple to use for the beginner. Available features, tweakable values and associated constraints are represented by **configuration tokens** and described by developers through **configuration description files** (.config) in the source tree.

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).

User point of view

Invocation

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
```

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 trunk/mutekh/examples/hello for examples of working build configuration files.

The MutekH API reference manual describes all available configuration tokens.

Help display

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
```

Module declaration

A build configuration file may declare a new module. 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 trunk/mutekh/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.

Developer point of view

MutekH has a component-based architecture where each module declares its configuration tokens.

Tokens are declared in configuration description files which are located at various places in the MutekH source tree. These constraints configuration files have a different syntax from the build configuration files. They are designed to declare configuration tokens and express relationships between available tokens.

Declared tokens can have their value changed in build configuration files and can be tested from C source code and Makefile.

The .config constraints files

Their are several types of configuration tokens:

- normal features enabling tokens which can be either defined or undefined in build configuration files.
- meta tokens which can only get defined through definition of other tokens.
- value tokens which can have any value.

Token flags

Several flags can be attached to tokens, most important ones are:

`value`
: Indicate the token is a value token. Value tokens can not have dependencies but can take values other than `defined` and `undefined`

`meta`
: Indicate the token is a meta token which may only be defined by an other token using the `provide` tag.

`auto`
: Indicate the token may be automatically defined to satisfy dependencies.

Other flags can be attached to tokens:

`harddep`
: Indicate the token can not be safely undefined due to a unsatisfied dependency.

`mandatory`
: Indicate the token can not be undefined at all. Useful to enforce requirements on other tokens, mainly for mandatory modules.

`root`
: Indicate the token has no parent.

`internal`
: Indicate the token is for internal use and can not be defined in build configuration file directly.

`noexport`
: Indicate the token should not be written out in generated files.

`private`
: Indicate the token can not be used with `parent`, `depend` or `provide` tag from an other `.config` file.

Constraint tags

For each configuration token, one may use the following tags:

`desc Description string without quotes`
: Short description about the token, multiple `desc` tags will be concatenated.

`flags FLAGS [?]`
: Set some flags with special meaning for the token (see above).

`parent TOKEN`
: Hierarchical dependency, it ensures all token with a parent gets silently undefined if the parent is undefined. This prevents options enabled by default to stay enabled if the parent is disabled; this way it avoids errors due to unneeded requirements. This is also used to hide irrelevant tokens from the help screen if the parent token is undefined.

`default value`
: Set the token default value. `defined` and `undefined` values act as booleans. default value is `undefined` if this line is omitted.

`module name [long name]`
: The feature token is associated with a module name. A module with the given name and the actual config file directory will be considered for building when the token gets defined.

The following tags may be used to specify features constraints:

`depend TOKEN [?]`
: The tag must be used to express feature dependencies, at least one of the given feature tokens is required. Unsatisfied dependency undefine the current token and emit a notice, unless flags modify this behavior.

`single TOKEN [?]`
: Same as depend with the additional constraint that only one of the given tokens may be defined.

`exclude TOKEN`
: Specify excluded tokens, the current token must not be defined at the same time as any given token.

`when TOKEN_CONDITION [?]`
: The current feature token will be automatically defined if all specified conditions are met. Missing dependencies will emit a notice as if it was defined in the build configuration file.

`provide TOKEN`
: Define a meta token if the current token is defined.

Some tags may be used to deals with values tokens. Value tokens must have the `value` flag set:

`require TOKEN_CONDITION [?]`

Requirements on value tokens, having at least one condition evaluates to true on the line is mandatory if the current token is defined.

`provide TOKEN=value`
: Set a value token to the specified value if the current token is defined.

Some tags can be used to give some configurations advice to the user when building MutekH:

`suggest TOKEN_CONDITION`
: Defining the current feature token suggest the given condition to the user.

`suggest_when TOKEN_CONDITION [?]`
: The current token will be suggested to the user if dependencies are actually satisfied and all given conditions are met.

The `TOKEN_CONDITION` might check different conditions:

- Token definition check: `TOKEN` or `TOKEN!`
- Token value equality check: `TOKEN=value`
- Token numerical value magnitude check: `TOKEN<value` or `TOKEN>value`

The configuration tool will check both constraint rules consistency and build configuration file respect of the rules when building MutekH.

Configuration constraints example:

```
%config CONFIG_FEATURE
  desc This is a great module for MutekH
  depend CONFIG_MUTEK_SCHEDULER
  module great The great library
  require CONFIG_CPU_MAXCOUNT>1
%config end

%config CONFIG_FEATURE_DEBUG
  desc Enable debug mode for the great feature
  parent CONFIG_FEATURE
  provide CONFIG_FEATURE_STACK_SIZE=4096
  when CONFIG_DEBUG
%config end

%config CONFIG_FEATURE_STACK_SIZE
  desc This is the thread stack size for the great feature
  parent CONFIG_FEATURE
  flags value
  default 512
%config end
```

Source tree Makefile syntax & rules

Makefiles in source directories may use the following variables:

`objs`
: A list of .o files compiled from `.c`, `.s` or `.S` files

`meta`
: A list of files that may be translated from `.m4`, `.cpp` or `.def` files

`copy`
: A list of files that must be copied verbatim from source directory to object directory

`subdirs`

A list of subdirectories where more files are to be processed. These directories must exist and contain a `Makefile`.

`doc_headers`
: A list of header files which must be parsed to generate the MutekH API reference manual, see header documentation for details.

Makefiles may contain optional flags that may be used for compilation:

`file.o_CFLAGS=?`
: CFLAGS to use for a given object

`DIR_CFLAGS=?`
: CFLAGS to use for all the objects compiled by the current Makefile. Flags added by this setting add-up with the object-specific ones above.

Moreover, one may use `ifeq (?,?)` make constructs to conditionally compile different things. Configuration tokens are usable.

Example:

```
objs = main.o

ifeq ($(CONFIG_SRL_SOCLIB),defined)
objs += barrier.o sched_wait.o srl_log.o hw_init.o
else
objs += posix_wait_cycles.o
endif

main.o_CFLAGS = -O0 -ggdb
```

The arch & cpu specific parts

Architecture and CPU directories have some special files which are injected in the building process:

- config.mk, included by make. It can define some compilation flags
- ldscript, invoked at link-time.
  - Architecture ldscript must create a loadable binary
  - CPU ldscript usually only specifies the entry point name

config.mk

The arch `config.mk` may override the following variables:

`ARCHCFLAGS`
: C-compiler flags

`ARCHLDFLAGS`
: Linker flags

`LD_NO_Q`
: Linker for the current architecture does not support -q switch, this slightly changes the linking process.

`HOSTCPPFLAGS`
: Flags to give to host's cpp (HOSTCPP) program. This is only used for expansion of `.def` files.

The cpu `config.mk` may override the following variables:

`CPUCFLAGS`

C-compiler flags

`CPULDFLAGS`
: Linker flags

ldscript

Try `info ld` for a guide through ldscripts?

This ldscript is taken from architecture's object directory, thus it may be obtained from either:

- copy
- m4 processing
- cpp processing

See arch/emu/ldscript, arch/soclib/ldscript.m4, and arch/simple/ldscript.cpp for the three flavors !

Notes

Prerequisites

The MutekH build-system is based on GNU Make features. It makes intensive use of:

- includes
- $(foreach) $(subst) $(eval) $(call) macros
- macro definitions

Therefore, a Make-3.81 at least is mandatory.