

Introduction

This document describes the MutekH build system configuration files and is intended for kernel developers.

Be sure to first read the [BuildSystem](#) page which contains more basic information.

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, express relationships between available tokens and describe associated constraints.

Declared tokens may be assigned in build configuration files to build with a given configuration. Their values can later be tested from source code and `Makefile` files using C macros and make variables.

The .config files syntax

Configuration description and constraints files contains blocks. Each block begins with `%config` or `%init` and declares a new token. See examples below for syntax details.

Configuration tokens declaration

There 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` CONFIG_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` CONFIG_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` CONFIG_TOKEN [?]

Same as `depend` with the additional constraint that only one of the given tokens may be defined.

`exclude` CONFIG_TOKEN

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

`when` CONFIG_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` CONFIG_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` CONFIG_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` CONFIG_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` CONFIG_TOKEN_CONDITION

Defining the current feature token suggest the given condition to the user.

suggest_when CONFIG_TOKEN_CONDITION [?]

The current token will be suggested to the user if dependencies are actually satisfied and all given conditions are met.

The CONFIG_TOKEN_CONDITION might check different conditions:

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

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

Example

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
```

Initialization tokens declaration

Initialization order of different software components at system start needs close attention. Having all modules and features initialized in proper order is challenging in modular and configurable projects like MutekH.

The configuration tools offers a way to specify initialization code ordering constraints and to optionally associate them to configuration tokens. This has several advantages:

- It allows inserting external modules initialization code in the right place without patching the main tree.
- It avoids the burdensome work of maintaining initialization function calls and associated `#ifdef` directives.
- It ensures initialization function invocations do not get badly reordered to satisfy a new constraint, while ignoring an older one.

A C source file will be generated with given initialization code respecting ordering constraints and current configuration. An error will be emitted if constraints can not be satisfied.

Constraint tags

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

parent CONFIG_TOKEN

When this tag is present, the initialization rules described in the block are ignored if the associated configuration token is undefined.

after INIT_TOKEN

This tag imposes the requirement that the code from current token be executed **after** the code associated with INIT_TOKEN.

before INIT_TOKEN

This tag imposes the requirement that the code from current token be executed **before** the code associated with INIT_TOKEN.

during INIT_TOKEN

This tag make token inherits from constraints expressed for the given INIT_TOKEN. The given token must be a place holder token and can not have associated code.

code C code here

This tag associates some initialization C code to the token. This tag may be used multiple times to add more code lines.

Example

Initialization constraints example:

```
%init INIT_FEATURE
  parent CONFIG_FEATURE
  during INIT_LIBRARIES
  after INIT_LIBC_STDIO      # an explanation why this is needed
  code great_feature_init();
%init end
```

Source tree Makefile syntax and 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/ and 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

The config.mk file

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

The ldscript file

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.

The configuration script requires perl ≥ 5.8 .