

This document describes the MutekH build system.

## Overview of the build process

The build system take a configuration file, processes the dependancies, and compiles the desired kernel.

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

The build system takes care of dependancies, file modifications, ...

## User point of view

### Makefile options (command line)

When building with 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
```

The following options are mandatory:

`CONF=`

An absolute path to the root configuration file for the desired kernel instance.

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.

`clean`

This 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.

```
$ make showconfig TOKEN=CONFIG_PTHREAD
```

## MutekH configuration files

MutekH configuration files contain tokens 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, ?)

Syntax is `token space value`. Tokens begin with `CONFIG_`, value may be unspecified 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

CONFIG_MUTEK_CONSOLE

# Device drivers
CONFIG_DRIVER_CHAR_EMUTTY

# Code compilation options
CONFIG_COMPILE_DEBUG
```

A configuration file may declare a new module, telling the build system the directory where the configuration lies has a local Makefile and some more objects to build.

```
# New source code module to be compiled
CONFIG_MODULES hello:%CONFIGPATH
```

For the list of all available tokens, do

```
$ make listallconfig
```

For a list of current available tokens depending on your configuration file, do

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

## Developer point of view

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

### The xxx.config files

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

```
desc Description string without quotes
    Short description about the token
parent TOKEN
    Parent token is help screen. This is only for pretty-printing.
require TOKEN [?]
    Mandatory requirements, having at least one of the tokens on the line is mandatory, conflict yields error
depend TOKEN [?]
    Dependencies, having at least one of the tokens on the line is mandatory, conflict implicitly undefines the
    current token
provide TOKEN [?]
    Mandatory forced requirements, given tokens are defined, conflict yields an error
provide TOKEN=value
    Mandatory forced requirements variant, sets a configuration token to a given value; value must not be
    undefined
provide TOKEN+=value
    Mandatory forced requirements variant, adds a value to a configuration token
exclude TOKEN
    Mandatory unrequirements, the specified token must not be defined
suggest TOKEN [?]
    Makes a token suggest other tokens when it is used. This is for help listing.
single TOKEN [?]
    Require a single one of the following tokens
```

Example:

```
%config CONFIG_SRL
desc MutekS API
provide CONFIG_MODULES+=libsrl:%CONFIGPATH
depend CONFIG_MUTEK_SCHEDULER
depend CONFIG_MWMMR
require CONFIG_SRL_SOCLIB CONFIG_SRL_STD
single CONFIG_SRL_SOCLIB CONFIG_SRL_STD
%config end
```

Here we declare a CONFIG\_SRL token

- needing CONFIG\_MUTEK\_SCHEDULER and CONFIG\_MWMMR,
- needing one of CONFIG\_SRL\_SOCLIB or CONFIG\_SRL\_STD,
- adding the directory containing the .conf as the "libsrl" module

## The directories Makefile syntax & rules

Makefiles in 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

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

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

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:

The directories Makefile syntax & rules

CPUCFLAGS

C-compiler flags

CPULDFLAGS

Linker flags

## Idscript

Try `info ld` for a guide through Idscripts?

This Idscript 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 base 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.