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.

`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.

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

Build configuration files

MutekH build 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 undefined

...
```

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

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:

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

To display 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 .config constraints files

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

`desc Description string without quotes`
: Short description about the token

`default value`
: Set the token default value. `defined` and `undefined` values act as booleans.

`depend TOKEN [?]`
: Dependencies, having at least one of the tokens on the line is required, if unsatisfied the current token gets undefined and a warning is emitted. May be used to disable features because of missing prerequisites.

`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 unrelevant tokens from the help screen if the parent token is undefined.

`require TOKEN [?]`
: Mandatory requirements, having at least one of the tokens on the line is mandatory, conflict yields error. May be usd to enforce definition of some mandatory configuration values.

`provide TOKEN [?]`
: Defining the current token enforce definition of the specified token.

`provide TOKEN=value`
: Defining the current token enforce definition of the specified token with the given value.

`provide TOKEN=+value`
: Defining the current token enforce definition of the specified token and concat the given value.

`exclude TOKEN`
: Mandatory excluded tokens, the specified token must not be defined

`suggest TOKEN [?]`
: Makes a token suggest other tokens when it is defined. This is for help listing.

`single TOKEN [?]`
: Only one of the following tokens may be defined at the same time

`fallback TOKEN`
: The fallback token will be enabled if the current one may not be enabled

The configuration tool may use multiple pass to find a valid configuration when tokens are disabled or enforced by given rules.

Example:

```
%module Module name for documentation

%config CONFIG_SRL
desc MutekS API
provide CONFIG_MODULES=+libsrl:$(CONFIGPATH)
depend CONFIG_MUTEK_SCHEDULER
depend CONFIG_MWMR
require CONFIG_SRL_SOCLIB CONFIG_SRL_STD
single CONFIG_SRL_SOCLIB CONFIG_SRL_STD
default undefined
%config end
```

Here we declare a `CONFIG_SRL` token

- needing CONFIG_MUTEK_SCHEDULER and CONFIG_MWMR,
- needing one of `CONFIG_SRL_SOCLIB` or `CONFIG_SRL_STD`,
- adding the directory containing the .conf as the "libsrl" module

Environment variable substitution takes place in both build and constraints configuration files.

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

`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.