The Build System of Athomux

Thomas Schöbel-Theuer

Version 0.07, 14 Dec 2004

Contents

1 Purpose

Athomux uses its own `make` system for building many different variants and configurations. The reason is manyfold: tools like `autoconf` and `automake` are taylored to userland, and Linux kernel configuration uses a *component software* paradigm. However, Athomux does not only exceed the component paradigm by a LEGO-like brickware paradigm, but also may be configured to run in different environments. Examples are guest environments running in Linux userland, or running in kernelspace, or running standalone. Therefore we need a build system capable of producing code for each of those environments.

The current build system is however not based on specifications of build environments or target architectures and the like, but rather allows description of *variants* for two phases of the build process: the Athomux preprocessor phase configuration (called `pconf` for short), and the C-compiler phase configuration (called `cconf`). For each of these phases, any number of variants may be specified. The *meaning* of a variant should be captured by its name.

To understand this document, you should have read some basic papers on the architecture of ATHOMUX, and you should be familiar with `Makefile`s (some knowledge of Perl may also help). Currently the description is very brief; you can help yourself by reading example code. A lot of stuff is missing; this document may soon be outdated.

2 Invocation

2.1 Invocation in General

`cd src; make`
Build all systems and all variants.

`cd src; make` *pconf_name/cconf_name/target_name*
Build a specific target (*target_name*) for a specific preprocessor configuration (*pconf_name*) and a specific C-compiler configuration (*cconf_name*).

2.2 Target specific Options

The ATHOMUX Linux kernelspace port requires a set of configured Linux kernel sources. If not explicit specified, the build system uses the sources of the current kernel. Different kernel sources can be specified by appending `LINSRC=`*/path/to/kernel/sources* to the `make` command. To specify *cconf* dependent kernel sources, the *cconf_name*`_LINSRC` option can be used.

To create Linux kernel modules, a kernel-dependent tool called `modpost` is required. Because of some portability and permission issues, the current build system uses a hack to avoid `modpost`. If you run into problems compiling the `athomux.ko` modules, you can reenable the usage of `modpost` by appending `USE_MODPOST=1` to the `make` command.

Example: `make klinux-i386 LINSRC=/usr/src/linux USE_MODPOST=1`

3 Configuration

3.1 Basic Config Files

3.1.1 `pconf`

By creating a file `pconf.`*`mypconfname`* in the `src` directory, the `make` system will notice that a new preprocessor configuration variant with name *`mypconfname`* exists. As a result, a directory named *`mypconfname`* will be automatically created as a subdirectory of `src`.

The file `pconf.`*`mypconfname`* can contain arbitray makefile rules, macro definitions, and so on, usually for invocation of the Athomux preprocessor `pre.pl`. These rules are included in the global Makefile via an automatically generated intermediate include-file `defs.make`. When you write different `pconf.*` files, all of their contents will be concatenated into `defs.make`, resulting in a single set of make rules. Thus make sure to avoid name clashes between different `pconf.*` versions.

As extension of ordinary `Makefile` rules, any occurence of `$(pconf)` in a `pconf.`*`mypconfname`* will be replaced by *`mypconfname`*. However notice that this replacement is done by the Perl script generating `defs.make`. Thus, it is possible to create parameterized macro names like `$(CFLAGS_$(pconf))` which will expand to $(CFLAGS_mypconfname). This way, you can separate name spaces of different versions.

It is highly recommended to do that with any kind of macros which could be different for different `pconf.*` versions.

3.1.2 `cconf`

By creating a file `cconf.`*`mycconfname`* in the `src` directory, the `make` system will be informed about the existence of a new C-configuration variant. As a result, a directory named *`mypconfname/mycconfname`* will be created automatically. By default, the full cartesian product of all `pconf.*` and `cconf.*` will be created.

A `cconf.*` can also contain arbitrary makefile rules, usually for invoking the C compiler.

Inside of a `cconf.`*`mycconfname`*, the pseudo-macros `$(pconf)` and `$(cconf)` can be used to denote the names of the current pconf and cconf variant, respectively. It is in particular recommended to parameterize makefile rules at least at the pconf level, because of the cartesian product with new `pconf.*` files which might be introduced at a later time.

3.1.3 `target`

By creating a file `target.`*`mytargetname`* in the `src` directory, the `make` system will be informed about the existence of a new `make` target. As a result, you can say `make` *`mypconfname/mycconfname/mytargetname`* for any combination of `pconf.*`, `cconf.*` and `target.*` (by default). When you just type `make` without any parameter, the full cartesian product of all pconfs, cconfs and targets will be built (by default).

Usually `target.*` will contain makefile rules for linking together an executable, configuring and building a bootable image or the like. As before, the pseudo-macros `$(pconf)` and `$(cconf)` can be used. Additionally, the pseudo-macro `$(target)` is available for parameterization of *`mytargetname`*.

3.1.4 Buildrules Embedded in `*.ath`

Further makefile rules can be added to `defs.make` by statements of the following form in a `*.ath` source file which must appear immediately *before* the `brick` statement:
`buildrules` *kind*`:` *makefile-rules-text*`....\n endrules`

where *kind* is one of the keywords `global`, `pconf`, `cconf`, or `target`.

In a `global` buildrule, no pseudo-macros are defined at all. In a `pconf` buildrule, only `$(pconf)` can be used. In a `cconf` buildrule, both `$(pconf)` and `$(cconf)` can be used. In a `target` buildrule, all three pseudo-macros including `$(target)` can be used.

Notice that depending on the *kind*, the number of copies of the *makefile-rules-text* may vary drastically. For `buildrules target:`, the full cartesian product of all pconfs, cconfs and targets will be generated and copied into `defs.make`. Please make sure that no name clashes can occur due to multiple unparameterized copies of the same *makefile-rules-text*.

Please try to prefer the `pconf.*`, `cconf.*` and `target.*` files in preference of `*.ath` buildrules. Only when some specific bricks (e.g. machine- or target-specific bricks) need additional makefile support, use `buildrules` statements.

The most common usage for `buildrules` is linking with external libraries, invocation of `make` on foreign source trees (e.g. foreign device drivers), and the like.

3.2 Filtering

The creation of the *full* cartesian product of all `pconf.*`, `cconf.*` and `target.*` can be restricted by filtering.

3.2.1 General Filters

In a `pconf.*`, `cconf.*`, `target.*` oder `*.ath` source file, you can add statements of the form
`#context pconf:` *regex-list*
`#context cconf:` *regex-list*
`#context target:` *regex-list*
`#context ath:` *regex-list*

(see also the Athomux Preprocessor Guide). The *regex-list* is a comma-separated list of Perl regular expressions, each of them potentially matching the *name* part of a `pconf.`*name*, `cconf.`*name*, `target.`*name*, or *name*`.ath` as a whole. When a regex is preceded by `!` (exclamation mark), the corresponding combination of the current source file with the matching source file will be excluded from, otherwise it will be included to the combinations which should be built. The rules are processed *in sequence*, such that later regexes will override the effects of earlier regexes.

HINT: if you want to exclude everything except a specific configuration, you can write a rule like `context pconf: !.*, ulinux` which first excludes all existing pconfs, and then selectively adds exactly `pconf.ulinux` to the combinations which should be built.

IMPORTANT: when you specifiy contradictory rules (e.g. in `target.A` you exclude `B.ath` while and in `B.ath` you include `target.A`), the following precedence rules apply: `pconf.*` < `cconf.*` < `target.*` < `*.ath`. A regex rule in a higher file will always supersede a rule from a lower one.

3.2.2 Generic Shell Filters

Some build problems depend on the machine where the build process is executed. For example, foreign architectures cannot be built on many architectures (exept you have cross-compilers etc). In order to limit the build configuration to the current capabilities of your system, the following context rules can also be used at any `pconf.*`, `cconf.*`, `target.*` and `*.ath`:

`# context cmd "`*shell-commands*`":` *list*

As you will expect, it calls the shell commands in Perl backquotes and checks whether the output of the command (after stripping the trailing newline) matches the *list* (positively or negatively as explained above).

As an example, you may check for a particular processor type by `#context cmd "uname -t": i386\n`.

3.3 Sub-Configurations

Often different `cconf.*` versions share a lot of common macro definitions or make rules. In order to remove redundancy, you are advised to put common things in include files. Whenever a `include` statement is found in one of the configuation files on a separate line, the inclusion is performed by `makegen.pl` such that the pseudo-macros valid at the calling file are also substituted in the included file.

This way, you can not only save redundancy, but also produce sub-configurations in a *systematic* way if you obey the following conventions:

`cconf-include.`*commonname* should denote a common include file for `cconf.`*commonname*`-subversion1` and `cconf.`*commonname*`-subversion2` and so on.

This means, you should produce subversions of configurations by means of hierarchical file names, where each hierarchy level is separated by dashes in the name. For example, if you want to discriminate different machine architectures for a common runtime environment type, you should create names like `cconf.klinux-i386` and `cconf.klinux-x86_64` with a common name part `klinux` and a common include file `cconf-include.klinux`.

This schema should be analogously extended to multiple hierarchy levels, e.g. when sub-versioning the `i386` architecture into `i386-pentium` and `i386-athlon` or the like. This way, you can create arbitrarily fine-grained hierarchical subversions of configurations (even with different numbers of hierarchy levels at different parts of the tree) without introducing redundancy, just by putting common parts into `cconf-include.`*shorter-version-name*.

In case you need a stronger binding between different parts of a name, you can use the dot instead of a dash for separating a hierarchical group as a whole from another group as a whole. Examples would be version names like `klinux-i386-athlon.debug` or even `klinux-i386-athlon.debug-gdb3` when different debuggers come into play (or the like). In such a case, please consider the use of multiple `include` statements for independent inclusion of independent things, in order to keep things as orthogonal as possible.

For `pconf`s, `target`s and `buildrules` sections of `*.ath` files you should use the analogous conventions `pconf-include.`*`commonname`*, `target-include.`*`commonname`* and `ath-include-`*`brickname-kind`*`.`*`commonname`* where *`commonname`* may itself be hierarchically structured.

4 Internals

The file `defs.make` is created by a Perl script `makegen.pl`, which is automatically invoked by the main `Makefile` whenever one of the `pconf.*`, `cconf.*`, `target.*` or `*.ath` is touched. Thus you should not usually have to bother with the internals.

5 TODO

A lot...

Probably the `*.ath` files should be organized in a hierarchy of subdirectories, by using the directory names as parts of the brick names (similar to Java libraries). Otherwise the management of hundrets or thousands of brick types could become a mess. Ideas for good systematics are sought.