Linux Kbuild Documentation

The kernel development community

Jun 10, 2024

CONTENTS

1 Kconfig Language 1

2 Kconfig macro language 15

3 Kbuild 21

4 Kconfig make config 29

5 Linux Kernel Makefiles 37

6 Building External Modules 65

7 Exporting kernel headers for use by userspace 75

8 Recursion issues 77

9 Reproducible builds 81

10 GCC plugin infrastructure 85

11 Building Linux with Clang/LLVM 89

CHAPTER

ONE

KCONFIG LANGUAGE

1.1 Introduction

The configuration database is a collection of configuration options organized in a tree structure:

```
+- Code maturity level options
|  +- Prompt for development and/or incomplete code/drivers
+- General setup
|  +- Networking support
|  +- System V IPC
|  +- BSD Process Accounting
|  +- Sysctl support
+- Loadable module support
|  +- Enable loadable module support
|     +- Set version information on all module symbols
|     +- Kernel module loader
+- ...
```

Every entry has its own dependencies. These dependencies are used to determine the visibility of an entry. Any child entry is only visible if its parent entry is also visible.

1.2 Menu entries

Most entries define a config option; all other entries help to organize them. A single configuration option is defined like this:

```
config MODVERSIONS
      bool "Set version information on all module symbols"
      depends on MODULES
      help
        Usually, modules have to be recompiled whenever you switch to a new
        kernel.  ...
```

Every line starts with a key word and can be followed by multiple arguments. “config” starts a new config entry. The following lines define attributes for this config option. Attributes can be the type of the config option, input prompt, dependencies, help text and default values. A config option can be defined multiple times with the same name, but every definition can have only a single input prompt and the type must not conflict.

1.3 Menu attributes

A menu entry can have a number of attributes. Not all of them are applicable everywhere (see syntax).

- type definition: “bool”/”tristate”/”string”/”hex”/”int”

  Every config option must have a type. There are only two basic types: tristate and string; the other types are based on these two. The type definition optionally accepts an input prompt, so these two examples are equivalent:

```
bool "Networking support"
```

  and:

```
bool
prompt "Networking support"
```

- input prompt: “prompt” <prompt> [“if” <expr>]

  Every menu entry can have at most one prompt, which is used to display to the user. Optionally dependencies only for this prompt can be added with “if”.

- default value: “default” <expr> [“if” <expr>]

  A config option can have any number of default values. If multiple default values are visible, only the first defined one is active. Default values are not limited to the menu entry where they are defined. This means the default can be defined somewhere else or be overridden by an earlier definition. The default value is only assigned to the config symbol if no other value was set by the user (via the input prompt above). If an input prompt is visible the default value is presented to the user and can be overridden by him. Optionally, dependencies only for this default value can be added with “if”.

  The default value deliberately defaults to ‘n’ in order to avoid bloating the build. With few exceptions, new config options should not change this. The intent is for “make oldconfig” to add as little as possible to the config from release to release.

  Note:
  Things that merit “default y/m” include:

  a) A new Kconfig option for something that used to always be built should be “default y”.

  b) A new gatekeeping Kconfig option that hides/shows other Kconfig options (but does not generate any code of its own), should be “default y” so people will see those other options.

  c) Sub-driver behavior or similar options for a driver that is “default n”. This allows you to provide sane defaults.

  d) Hardware or infrastructure that everybody expects, such as CONFIG_NET or CONFIG_BLOCK. These are rare exceptions.

- type definition + default value:

```
"def_bool"/"def_tristate" <expr> ["if" <expr>]
```

This is a shorthand notation for a type definition plus a value. Optionally dependencies for this default value can be added with “if”.

- dependencies: “depends on” <expr>

  This defines a dependency for this menu entry. If multiple dependencies are defined, they are connected with ‘&&’. Dependencies are applied to all other options within this menu entry (which also accept an “if” expression), so these two examples are equivalent:

```
bool "foo" if BAR
default y if BAR
```

  and:

```
depends on BAR
bool "foo"
default y
```

- reverse dependencies: “select” <symbol> [“if” <expr>]

  While normal dependencies reduce the upper limit of a symbol (see below), reverse dependencies can be used to force a lower limit of another symbol. The value of the current menu symbol is used as the minimal value <symbol> can be set to. If <symbol> is selected multiple times, the limit is set to the largest selection. Reverse dependencies can only be used with boolean or tristate symbols.

  Note:
  select should be used with care. select will force a symbol to a value without visiting the dependencies. By abusing select you are able to select a symbol FOO even if FOO depends on BAR that is not set. In general use select only for non-visible symbols (no prompts anywhere) and for symbols with no dependencies. That will limit the usefulness but on the other hand avoid the illegal configurations all over.

- weak reverse dependencies: “imply” <symbol> [“if” <expr>]

  This is similar to “select” as it enforces a lower limit on another symbol except that the “implied” symbol’s value may still be set to n from a direct dependency or with a visible prompt.

  Given the following example:

```
config FOO
    tristate "foo"
    imply BAZ

config BAZ
    tristate "baz"
    depends on BAR
```

  The following values are possible:

FOO	BAR	BAZ's default	choice for BAZ
n	y	n	N/m/y
m	y	m	M/y/n
y	y	y	Y/m/n
n	m	n	N/m
m	m	m	M/n
y	m	m	M/n
y	n	-	N

This is useful e.g. with multiple drivers that want to indicate their ability to hook into a secondary subsystem while allowing the user to configure that subsystem out without also having to unset these drivers.

Note: If the combination of FOO=y and BAR=m causes a link error, you can guard the function call with IS_REACHABLE():

```
foo_init()
{
        if (IS_REACHABLE(CONFIG_BAZ))
                baz_register(&foo);
        ...
}
```

Note: If the feature provided by BAZ is highly desirable for FOO, FOO should imply not only BAZ, but also its dependency BAR:

```
config FOO
    tristate "foo"
    imply BAR
    imply BAZ
```

- limiting menu display: "visible if" <expr>

  This attribute is only applicable to menu blocks, if the condition is false, the menu block is not displayed to the user (the symbols contained there can still be selected by other symbols, though). It is similar to a conditional "prompt" attribute for individual menu entries. Default value of "visible" is true.

- numerical ranges: "range" <symbol> <symbol> ["if" <expr>]

  This allows to limit the range of possible input values for int and hex symbols. The user can only input a value which is larger than or equal to the first symbol and smaller than or equal to the second symbol.

- help text: "help"

  This defines a help text. The end of the help text is determined by the indentation level, this means it ends at the first line which has a smaller indentation than the first line of the help text.

- module attribute: "modules" This declares the symbol to be used as the MODULES symbol, which enables the third modular state for all config symbols. At most one symbol may have

the “modules” option set.

1.4 Menu dependencies

Dependencies define the visibility of a menu entry and can also reduce the input range of tristate symbols. The tristate logic used in the expressions uses one more state than normal boolean logic to express the module state. Dependency expressions have the following syntax:

```
<expr> ::= <symbol>                             (1)
         <symbol> '=' <symbol>                  (2)
         <symbol> '!=' <symbol>                 (3)
         <symbol1> '<' <symbol2>                (4)
         <symbol1> '>' <symbol2>                (4)
         <symbol1> '<=' <symbol2>               (4)
         <symbol1> '>=' <symbol2>               (4)
         '(' <expr> ')'                         (5)
         '!' <expr>                             (6)
         <expr> '&&' <expr>                     (7)
         <expr> '||' <expr>                     (8)
```

Expressions are listed in decreasing order of precedence.

(1) Convert the symbol into an expression. Boolean and tristate symbols are simply converted into the respective expression values. All other symbol types result in ‘n’.

(2) If the values of both symbols are equal, it returns ‘y’, otherwise ‘n’.

(3) If the values of both symbols are equal, it returns ‘n’, otherwise ‘y’.

(4) If value of <symbol1> is respectively lower, greater, lower-or-equal, or greater-or-equal than value of <symbol2>, it returns ‘y’, otherwise ‘n’.

(5) Returns the value of the expression. Used to override precedence.

(6) Returns the result of (2-/expr/).

(7) Returns the result of min(/expr/, /expr/).

(8) Returns the result of max(/expr/, /expr/).

An expression can have a value of ‘n’, ‘m’ or ‘y’ (or 0, 1, 2 respectively for calculations). A menu entry becomes visible when its expression evaluates to ‘m’ or ‘y’.

There are two types of symbols: constant and non-constant symbols. Non-constant symbols are the most common ones and are defined with the ‘config’ statement. Non-constant symbols consist entirely of alphanumeric characters or underscores. Constant symbols are only part of expressions. Constant symbols are always surrounded by single or double quotes. Within the quote, any other character is allowed and the quotes can be escaped using ‘\’.

1.5 Menu structure

The position of a menu entry in the tree is determined in two ways. First it can be specified explicitly:

```
menu "Network device support"
      depends on NET

config NETDEVICES
      ...

endmenu
```

All entries within the “menu” ... “endmenu” block become a submenu of “Network device support”. All subentries inherit the dependencies from the menu entry, e.g. this means the dependency “NET” is added to the dependency list of the config option NETDEVICES.

The other way to generate the menu structure is done by analyzing the dependencies. If a menu entry somehow depends on the previous entry, it can be made a submenu of it. First, the previous (parent) symbol must be part of the dependency list and then one of these two conditions must be true:

- the child entry must become invisible, if the parent is set to ‘n’
- the child entry must only be visible, if the parent is visible:

```
config MODULES
    bool "Enable loadable module support"

config MODVERSIONS
    bool "Set version information on all module symbols"
    depends on MODULES

comment "module support disabled"
    depends on !MODULES
```

MODVERSIONS directly depends on MODULES, this means it’s only visible if MODULES is different from ‘n’. The comment on the other hand is only visible when MODULES is set to ‘n’.

1.6 Kconfig syntax

The configuration file describes a series of menu entries, where every line starts with a keyword (except help texts). The following keywords end a menu entry:

- config
- menuconfig
- choice/endchoice
- comment
- menu/endmenu

- if/endif
- source

The first five also start the definition of a menu entry.

config:

```
"config" <symbol>
<config options>
```

This defines a config symbol <symbol> and accepts any of above attributes as options.

menuconfig:

```
"menuconfig" <symbol>
<config options>
```

This is similar to the simple config entry above, but it also gives a hint to front ends, that all suboptions should be displayed as a separate list of options. To make sure all the suboptions will really show up under the menuconfig entry and not outside of it, every item from the <config options> list must depend on the menuconfig symbol. In practice, this is achieved by using one of the next two constructs:

```
(1):
menuconfig M
if M
    config C1
    config C2
endif

(2):
menuconfig M
config C1
    depends on M
config C2
    depends on M
```

In the following examples (3) and (4), C1 and C2 still have the M dependency, but will not appear under menuconfig M anymore, because of C0, which doesn't depend on M:

```
(3):
menuconfig M
    config C0
if M
    config C1
    config C2
endif

(4):
menuconfig M
config C0
config C1
    depends on M
```

```
config C2
    depends on M
```

choices:

```
"choice" [symbol]
<choice options>
<choice block>
"endchoice"
```

This defines a choice group and accepts any of the above attributes as options. A choice can only be of type bool or tristate. If no type is specified for a choice, its type will be determined by the type of the first choice element in the group or remain unknown if none of the choice elements have a type specified, as well.

While a boolean choice only allows a single config entry to be selected, a tristate choice also allows any number of config entries to be set to 'm'. This can be used if multiple drivers for a single hardware exists and only a single driver can be compiled/loaded into the kernel, but all drivers can be compiled as modules.

A choice accepts another option "optional", which allows to set the choice to 'n' and no entry needs to be selected. If no [symbol] is associated with a choice, then you can not have multiple definitions of that choice. If a [symbol] is associated to the choice, then you may define the same choice (i.e. with the same entries) in another place.

comment:

```
"comment" <prompt>
<comment options>
```

This defines a comment which is displayed to the user during the configuration process and is also echoed to the output files. The only possible options are dependencies.

menu:

```
"menu" <prompt>
<menu options>
<menu block>
"endmenu"
```

This defines a menu block, see "Menu structure" above for more information. The only possible options are dependencies and "visible" attributes.

if:

```
"if" <expr>
<if block>
"endif"
```

This defines an if block. The dependency expression <expr> is appended to all enclosed menu entries.

source:

```
"source" <prompt>
```

This reads the specified configuration file. This file is always parsed.

mainmenu:

```
"mainmenu" <prompt>
```

This sets the config program's title bar if the config program chooses to use it. It should be placed at the top of the configuration, before any other statement.

'#' Kconfig source file comment:

An unquoted '#' character anywhere in a source file line indicates the beginning of a source file comment. The remainder of that line is a comment.

1.7 Kconfig hints

This is a collection of Kconfig tips, most of which aren't obvious at first glance and most of which have become idioms in several Kconfig files.

1.7.1 Adding common features and make the usage configurable

It is a common idiom to implement a feature/functionality that are relevant for some architectures but not all. The recommended way to do so is to use a config variable named HAVE_* that is defined in a common Kconfig file and selected by the relevant architectures. An example is the generic IOMAP functionality.

We would in lib/Kconfig see:

```
# Generic IOMAP is used to ...
config HAVE_GENERIC_IOMAP

config GENERIC_IOMAP
      depends on HAVE_GENERIC_IOMAP && FOO
```

And in lib/Makefile we would see:

```
obj-$(CONFIG_GENERIC_IOMAP) += iomap.o
```

For each architecture using the generic IOMAP functionality we would see:

```
config X86
      select ...
      select HAVE_GENERIC_IOMAP
      select ...
```

Note: we use the existing config option and avoid creating a new config variable to select HAVE_GENERIC_IOMAP.

Note: the use of the internal config variable HAVE_GENERIC_IOMAP, it is introduced to overcome the limitation of select which will force a config option to 'y' no matter the dependencies.

The dependencies are moved to the symbol GENERIC_IOMAP and we avoid the situation where select forces a symbol equals to ‘y’.

1.7.2 Adding features that need compiler support

There are several features that need compiler support. The recommended way to describe the dependency on the compiler feature is to use “depends on” followed by a test macro:

```
config STACKPROTECTOR
      bool "Stack Protector buffer overflow detection"
      depends on $(cc-option,-fstack-protector)
      ...
```

If you need to expose a compiler capability to makefiles and/or C source files, *CC_HAS_* is the recommended prefix for the config option:

```
config CC_HAS_FOO
      def_bool $(success,$(srctree)/scripts/cc-check-foo.sh $(CC))
```

1.7.3 Build as module only

To restrict a component build to module-only, qualify its config symbol with “depends on m”. E.g.:

```
config FOO
      depends on BAR && m
```

limits FOO to module (=m) or disabled (=n).

1.7.4 Compile-testing

If a config symbol has a dependency, but the code controlled by the config symbol can still be compiled if the dependency is not met, it is encouraged to increase build coverage by adding an “|| COMPILE_TEST” clause to the dependency. This is especially useful for drivers for more exotic hardware, as it allows continuous-integration systems to compile-test the code on a more common system, and detect bugs that way. Note that compile-tested code should avoid crashing when run on a system where the dependency is not met.

1.7.5 Architecture and platform dependencies

Due to the presence of stubs, most drivers can now be compiled on most architectures. However, this does not mean it makes sense to have all drivers available everywhere, as the actual hardware may only exist on specific architectures and platforms. This is especially true for on-SoC IP cores, which may be limited to a specific vendor or SoC family.

To prevent asking the user about drivers that cannot be used on the system(s) the user is compiling a kernel for, and if it makes sense, config symbols controlling the compilation of a driver should contain proper dependencies, limiting the visibility of the symbol to (a superset of) the platform(s) the driver can be used on. The dependency can be an architecture (e.g. ARM) or

platform (e.g. ARCH_OMAP4) dependency. This makes life simpler not only for distro config owners, but also for every single developer or user who configures a kernel.

Such a dependency can be relaxed by combining it with the compile-testing rule above, leading to:

config FOO
bool "Support for foo hardware" depends on ARCH_FOO_VENDOR || COMPILE_TEST

1.7.6 Optional dependencies

Some drivers are able to optionally use a feature from another module or build cleanly with that module disabled, but cause a link failure when trying to use that loadable module from a built-in driver.

The most common way to express this optional dependency in Kconfig logic uses the slightly counterintuitive:

```
config FOO
      tristate "Support for foo hardware"
      depends on BAR || !BAR
```

This means that there is either a dependency on BAR that disallows the combination of FOO=y with BAR=m, or BAR is completely disabled. For a more formalized approach if there are multiple drivers that have the same dependency, a helper symbol can be used, like:

```
config FOO
      tristate "Support for foo hardware"
      depends on BAR_OPTIONAL

config BAR_OPTIONAL
      def_tristate BAR || !BAR
```

1.7.7 Kconfig recursive dependency limitations

If you've hit the Kconfig error: "recursive dependency detected" you've run into a recursive dependency issue with Kconfig, a recursive dependency can be summarized as a circular dependency. The kconfig tools need to ensure that Kconfig files comply with specified configuration requirements. In order to do that kconfig must determine the values that are possible for all Kconfig symbols, this is currently not possible if there is a circular relation between two or more Kconfig symbols. For more details refer to the "Simple Kconfig recursive issue" subsection below. Kconfig does not do recursive dependency resolution; this has a few implications for Kconfig file writers. We'll first explain why this issues exists and then provide an example technical limitation which this brings upon Kconfig developers. Eager developers wishing to try to address this limitation should read the next subsections.

1.7.8 Simple Kconfig recursive issue

Read: Documentation/kbuild/Kconfig.recursion-issue-01

Test with:

```
make KBUILD_KCONFIG=Documentation/kbuild/Kconfig.recursion-issue-01 allnoconfig
```

1.7.9 Cumulative Kconfig recursive issue

Read: Documentation/kbuild/Kconfig.recursion-issue-02

Test with:

```
make KBUILD_KCONFIG=Documentation/kbuild/Kconfig.recursion-issue-02 allnoconfig
```

1.7.10 Practical solutions to kconfig recursive issue

Developers who run into the recursive Kconfig issue have two options at their disposal. We document them below and also provide a list of historical issues resolved through these different solutions.

a) Remove any superfluous “select FOO” or “depends on FOO”

b) Match dependency semantics:

   b1) Swap all “select FOO” to “depends on FOO” or,

   b2) Swap all “depends on FOO” to “select FOO”

The resolution to a) can be tested with the sample Kconfig file Documentation/kbuild/Kconfig.recursion-issue-01 through the removal of the “select CORE” from CORE_BELL_A_ADVANCED as that is implicit already since CORE_BELL_A depends on CORE. At times it may not be possible to remove some dependency criteria, for such cases you can work with solution b).

The two different resolutions for b) can be tested in the sample Kconfig file Documentation/kbuild/Kconfig.recursion-issue-02.

Below is a list of examples of prior fixes for these types of recursive issues; all errors appear to involve one or more “select” statements and one or more “depends on”.

commit	fix
06b718c01208	select A -> depends on A
c22eacfe82f9	depends on A -> depends on B
6a91e854442c	select A -> depends on A
118c565a8f2e	select A -> select B
f004e5594705	select A -> depends on A
c7861f37b4c6	depends on A -> (null)
80c69915e5fb	select A -> (null) (1)
c2218e26c0d0	select A -> depends on A (1)
d6ae99d04e1c	select A -> depends on A
95ca19cf8cbf	select A -> depends on A
8f057d7bca54	depends on A -> (null)
8f057d7bca54	depends on A -> select A
a0701f04846e	select A -> depends on A
0c8b92f7f259	depends on A -> (null)
e4e9e0540928	select A -> depends on A (2)
7453ea886e87	depends on A > (null) (1)
7b1fff7e4fdf	select A -> depends on A
86c747d2a4f0	select A -> depends on A
d9f9ab51e55e	select A -> depends on A
0c51a4d8abd6	depends on A -> select A (3)
e98062ed6dc4	select A -> depends on A (3)
91e5d284a7f1	select A -> (null)

(1) Partial (or no) quote of error.

(2) That seems to be the gist of that fix.

(3) Same error.

1.7.11 Future kconfig work

Work on kconfig is welcomed on both areas of clarifying semantics and on evaluating the use of a full SAT solver for it. A full SAT solver can be desirable to enable more complex dependency mappings and / or queries, for instance one possible use case for a SAT solver could be that of handling the current known recursive dependency issues. It is not known if this would address such issues but such evaluation is desirable. If support for a full SAT solver proves too complex or that it cannot address recursive dependency issues Kconfig should have at least clear and well defined semantics which also addresses and documents limitations or requirements such as the ones dealing with recursive dependencies.

Further work on both of these areas is welcomed on Kconfig. We elaborate on both of these in the next two subsections.

1.7.12 Semantics of Kconfig

The use of Kconfig is broad, Linux is now only one of Kconfig's users: one study has completed a broad analysis of Kconfig use in 12 projects[0]. Despite its widespread use, and although this document does a reasonable job in documenting basic Kconfig syntax a more precise definition of Kconfig semantics is welcomed. One project deduced Kconfig semantics through the use of the xconfig configurator[1]. Work should be done to confirm if the deduced semantics matches our intended Kconfig design goals. Another project formalized a denotational semantics of a core subset of the Kconfig language[10].

Having well defined semantics can be useful for tools for practical evaluation of dependencies, for instance one such case was work to express in boolean abstraction of the inferred semantics of Kconfig to translate Kconfig logic into boolean formulas and run a SAT solver on this to find dead code / features (always inactive), 114 dead features were found in Linux using this methodology[1] (Section 8: Threats to validity). The kismet tool, based on the semantics in[10], finds abuses of reverse dependencies and has led to dozens of committed fixes to Linux Kconfig files[11].

Confirming this could prove useful as Kconfig stands as one of the leading industrial variability modeling languages[12]. Its study would help evaluate practical uses of such languages, their use was only theoretical and real world requirements were not well understood. As it stands though only reverse engineering techniques have been used to deduce semantics from variability modeling languages such as Kconfig[3].

1.7.13 Full SAT solver for Kconfig

Although SAT solvers[4] haven't yet been used by Kconfig directly, as noted in the previous subsection, work has been done however to express in boolean abstraction the inferred semantics of Kconfig to translate Kconfig logic into boolean formulas and run a SAT solver on it[5]. Another known related project is CADOS[6] (former VAMOS[7]) and the tools, mainly undertaker[8], which has been introduced first with[9]. The basic concept of undertaker is to extract variability models from Kconfig and put them together with a propositional formula extracted from CPP #ifdefs and build-rules into a SAT solver in order to find dead code, dead files, and dead symbols. If using a SAT solver is desirable on Kconfig one approach would be to evaluate repurposing such efforts somehow on Kconfig. There is enough interest from mentors of existing projects to not only help advise how to integrate this work upstream but also help maintain it long term. Interested developers should visit:

https://kernelnewbies.org/KernelProjects/kconfig-sat

[0] https://www.eng.uwaterloo.ca/~shshe/kconfig_semantics.pdf
[1] https://gsd.uwaterloo.ca/sites/default/files/vm-2013-berger.pdf
[10] https://paulgazzillo.com/papers/esecfse21.pdf
[11] https://github.com/paulgazz/kmax
[2] https://gsd.uwaterloo.ca/sites/default/files/ase241-berger_0.pdf
[3] https://gsd.uwaterloo.ca/sites/default/files/icse2011.pdf
[4] https://www.cs.cornell.edu/~sabhar/chapters/SATSolvers-KR-Handbook.pdf
[5] https://gsd.uwaterloo.ca/sites/default/files/vm-2013-berger.pdf
[6] https://cados.cs.fau.de
[7] https://vamos.cs.fau.de
[8] https://undertaker.cs.fau.de
[9] https://www4.cs.fau.de/Publications/2011/tartler_11_eurosys.pdf

CHAPTER

TWO

KCONFIG MACRO LANGUAGE

2.1 Concept

The basic idea was inspired by Make. When we look at Make, we notice sort of two languages in one. One language describes dependency graphs consisting of targets and prerequisites. The other is a macro language for performing textual substitution.

There is clear distinction between the two language stages. For example, you can write a makefile like follows:

```
APP := foo
SRC := foo.c
CC := gcc

$(APP): $(SRC)
        $(CC) -o $(APP) $(SRC)
```

The macro language replaces the variable references with their expanded form, and handles as if the source file were input like follows:

```
foo: foo.c
        gcc -o foo foo.c
```

Then, Make analyzes the dependency graph and determines the targets to be updated.

The idea is quite similar in Kconfig - it is possible to describe a Kconfig file like this:

```
CC := gcc

config CC_HAS_FOO
        def_bool $(shell, $(srctree)/scripts/gcc-check-foo.sh $(CC))
```

The macro language in Kconfig processes the source file into the following intermediate:

```
config CC_HAS_FOO
        def_bool y
```

Then, Kconfig moves onto the evaluation stage to resolve inter-symbol dependency as explained in *Kconfig Language*.

2.2 Variables

Like in Make, a variable in Kconfig works as a macro variable. A macro variable is expanded "in place" to yield a text string that may then be expanded further. To get the value of a variable, enclose the variable name in $( ). The parentheses are required even for single-letter variable names; $X is a syntax error. The curly brace form as in ${CC} is not supported either.

There are two types of variables: simply expanded variables and recursively expanded variables.

A simply expanded variable is defined using the := assignment operator. Its righthand side is expanded immediately upon reading the line from the Kconfig file.

A recursively expanded variable is defined using the = assignment operator. Its righthand side is simply stored as the value of the variable without expanding it in any way. Instead, the expansion is performed when the variable is used.

There is another type of assignment operator; += is used to append text to a variable. The righthand side of += is expanded immediately if the lefthand side was originally defined as a simple variable. Otherwise, its evaluation is deferred.

The variable reference can take parameters, in the following form:

```
$(name,arg1,arg2,arg3)
```

You can consider the parameterized reference as a function. (more precisely, "user-defined function" in contrast to "built-in function" listed below).

Useful functions must be expanded when they are used since the same function is expanded differently if different parameters are passed. Hence, a user-defined function is defined using the = assignment operator. The parameters are referenced within the body definition with $(1), $(2), etc.

In fact, recursively expanded variables and user-defined functions are the same internally. (In other words, "variable" is "function with zero argument".) When we say "variable" in a broad sense, it includes "user-defined function".

2.3 Built-in functions

Like Make, Kconfig provides several built-in functions. Every function takes a particular number of arguments.

In Make, every built-in function takes at least one argument. Kconfig allows zero argument for built-in functions, such as $(filename), $(lineno). You could consider those as "built-in variable", but it is just a matter of how we call it after all. Let's say "built-in function" here to refer to natively supported functionality.

Kconfig currently supports the following built-in functions.

- $(shell,command)

  The "shell" function accepts a single argument that is expanded and passed to a subshell for execution. The standard output of the command is then read and returned as the value of the function. Every newline in the output

is replaced with a space. Any trailing newlines are deleted. The standard error is not returned, nor is any program exit status.

- $(info,text)

  The “info” function takes a single argument and prints it to stdout. It evaluates to an empty string.

- $(warning-if,condition,text)

  The “warning-if” function takes two arguments. If the condition part is “y”, the text part is sent to stderr. The text is prefixed with the name of the current Kconfig file and the current line number.

- $(error-if,condition,text)

  The “error-if” function is similar to “warning-if”, but it terminates the parsing immediately if the condition part is “y”.

- $(filename)

  The ‘filename’ takes no argument, and $(filename) is expanded to the file name being parsed.

- $(lineno)

  The ‘lineno’ takes no argument, and $(lineno) is expanded to the line number being parsed.

2.4 Make vs Kconfig

Kconfig adopts Make-like macro language, but the function call syntax is slightly different.

A function call in Make looks like this:

```
$(func-name arg1,arg2,arg3)
```

The function name and the first argument are separated by at least one whitespace. Then, leading whitespaces are trimmed from the first argument, while whitespaces in the other arguments are kept. You need to use a kind of trick to start the first parameter with spaces. For example, if you want to make “info” function print “ hello”, you can write like follows:

```
empty :=
space := $(empty) $(empty)
$(info $(space)$(space)hello)
```

Kconfig uses only commas for delimiters, and keeps all whitespaces in the function call. Some people prefer putting a space after each comma delimiter:

```
$(func-name, arg1, arg2, arg3)
```

In this case, “func-name” will receive “ arg1”, “ arg2”, “ arg3”. The presence of leading spaces may matter depending on the function. The same applies to Make - for example, $(subst .c, .o, $(sources)) is a typical mistake; it replaces “.c” with “ .o”.

In Make, a user-defined function is referenced by using a built-in function, ‘call’, like this:

```
$(call my-func,arg1,arg2,arg3)
```

Kconfig invokes user-defined functions and built-in functions in the same way. The omission of 'call' makes the syntax shorter.

In Make, some functions treat commas verbatim instead of argument separators. For example, $(shell echo hello, world) runs the command "echo hello, world". Likewise, $(info hello, world) prints "hello, world" to stdout. You could say this is _useful_ inconsistency.

In Kconfig, for simpler implementation and grammatical consistency, commas that appear in the $( ) context are always delimiters. It means:

```
$(shell, echo hello, world)
```

is an error because it is passing two parameters where the 'shell' function accepts only one. To pass commas in arguments, you can use the following trick:

```
comma := ,
$(shell, echo hello$(comma) world)
```

2.5 Caveats

A variable (or function) cannot be expanded across tokens. So, you cannot use a variable as a shorthand for an expression that consists of multiple tokens. The following works:

```
RANGE_MIN := 1
RANGE_MAX := 3

config FOO
        int "foo"
        range $(RANGE_MIN) $(RANGE_MAX)
```

But, the following does not work:

```
RANGES := 1 3

config FOO
        int "foo"
        range $(RANGES)
```

A variable cannot be expanded to any keyword in Kconfig. The following does not work:

```
MY_TYPE := tristate

config FOO
        $(MY_TYPE) "foo"
        default y
```

Obviously from the design, $(shell command) is expanded in the textual substitution phase. You cannot pass symbols to the 'shell' function.

The following does not work as expected:

```
config ENDIAN_FLAG
        string
        default "-mbig-endian" if CPU_BIG_ENDIAN
        default "-mlittle-endian" if CPU_LITTLE_ENDIAN

config CC_HAS_ENDIAN_FLAG
        def_bool $(shell $(srctree)/scripts/gcc-check-flag ENDIAN_FLAG)
```

Instead, you can do like follows so that any function call is statically expanded:

```
config CC_HAS_ENDIAN_FLAG
        bool
        default $(shell $(srctree)/scripts/gcc-check-flag -mbig-endian) if CPU_
↪BIG_ENDIAN
        default $(shell $(srctree)/scripts/gcc-check-flag -mlittle-endian) if␣
↪CPU_LITTLE_ENDIAN
```

CHAPTER

THREE

KBUILD

3.1 Output files

3.1.1 modules.order

This file records the order in which modules appear in Makefiles. This is used by modprobe to deterministically resolve aliases that match multiple modules.

3.1.2 modules.builtin

This file lists all modules that are built into the kernel. This is used by modprobe to not fail when trying to load something builtin.

3.1.3 modules.builtin.modinfo

This file contains modinfo from all modules that are built into the kernel. Unlike modinfo of a separate module, all fields are prefixed with module name.

3.2 Environment variables

3.2.1 KCPPFLAGS

Additional options to pass when preprocessing. The preprocessing options will be used in all cases where kbuild does preprocessing including building C files and assembler files.

3.2.2 KAFLAGS

Additional options to the assembler (for built-in and modules).

3.2.3 AFLAGS_MODULE

Additional assembler options for modules.

3.2.4 AFLAGS_KERNEL

Additional assembler options for built-in.

3.2.5 KCFLAGS

Additional options to the C compiler (for built-in and modules).

3.2.6 KRUSTFLAGS

Additional options to the Rust compiler (for built-in and modules).

3.2.7 CFLAGS_KERNEL

Additional options for $(CC) when used to compile code that is compiled as built-in.

3.2.8 CFLAGS_MODULE

Additional module specific options to use for $(CC).

3.2.9 RUSTFLAGS_KERNEL

Additional options for $(RUSTC) when used to compile code that is compiled as built-in.

3.2.10 RUSTFLAGS_MODULE

Additional module specific options to use for $(RUSTC).

3.2.11 LDFLAGS_MODULE

Additional options used for $(LD) when linking modules.

3.2.12 HOSTCFLAGS

Additional flags to be passed to $(HOSTCC) when building host programs.

3.2.13 HOSTCXXFLAGS

Additional flags to be passed to $(HOSTCXX) when building host programs.

3.2.14 HOSTRUSTFLAGS

Additional flags to be passed to $(HOSTRUSTC) when building host programs.

3.2.15 HOSTLDFLAGS

Additional flags to be passed when linking host programs.

3.2.16 HOSTLDLIBS

Additional libraries to link against when building host programs.

3.2.17 USERCFLAGS

Additional options used for $(CC) when compiling userprogs.

3.2.18 USERLDFLAGS

Additional options used for $(LD) when linking userprogs. userprogs are linked with CC, so $(USERLDFLAGS) should include "-Wl," prefix as applicable.

3.2.19 KBUILD_KCONFIG

Set the top-level Kconfig file to the value of this environment variable. The default name is "Kconfig".

3.2.20 KBUILD_VERBOSE

Set the kbuild verbosity. Can be assigned same values as "V=...".

See make help for the full list.

Setting "V=..." takes precedence over KBUILD_VERBOSE.

3.2.21 KBUILD_EXTMOD

Set the directory to look for the kernel source when building external modules.

Setting “M=...” takes precedence over KBUILD_EXTMOD.

3.2.22 KBUILD_OUTPUT

Specify the output directory when building the kernel.

The output directory can also be specified using “O=...”.

Setting “O=...” takes precedence over KBUILD_OUTPUT.

3.2.23 KBUILD_EXTRA_WARN

Specify the extra build checks. The same value can be assigned by passing W=... from the command line.

See *make help* for the list of the supported values.

Setting “W=...” takes precedence over KBUILD_EXTRA_WARN.

3.2.24 KBUILD_DEBARCH

For the deb-pkg target, allows overriding the normal heuristics deployed by deb-pkg. Normally deb-pkg attempts to guess the right architecture based on the UTS_MACHINE variable, and on some architectures also the kernel config. The value of KBUILD_DEBARCH is assumed (not checked) to be a valid Debian architecture.

3.2.25 KDOCFLAGS

Specify extra (warning/error) flags for kernel-doc checks during the build, see scripts/kernel-doc for which flags are supported. Note that this doesn’t (currently) apply to documentation builds.

3.2.26 ARCH

Set ARCH to the architecture to be built.

In most cases the name of the architecture is the same as the directory name found in the arch/ directory.

But some architectures such as x86 and sparc have aliases.

- x86: i386 for 32 bit, x86_64 for 64 bit
- parisc: parisc64 for 64 bit
- sparc: sparc32 for 32 bit, sparc64 for 64 bit

3.2.27 CROSS_COMPILE

Specify an optional fixed part of the binutils filename. CROSS_COMPILE can be a part of the filename or the full path.

CROSS_COMPILE is also used for ccache in some setups.

3.2.28 CF

Additional options for sparse.

CF is often used on the command-line like this:

```
make CF=-Wbitwise C=2
```

3.2.29 INSTALL_PATH

INSTALL_PATH specifies where to place the updated kernel and system map images. Default is /boot, but you can set it to other values.

3.2.30 INSTALLKERNEL

Install script called when using “make install”. The default name is “installkernel”.

The script will be called with the following arguments:

- $1 - kernel version
- $2 - kernel image file
- $3 - kernel map file
- $4 - default install path (use root directory if blank)

The implementation of “make install” is architecture specific and it may differ from the above.

INSTALLKERNEL is provided to enable the possibility to specify a custom installer when cross compiling a kernel.

3.2.31 MODLIB

Specify where to install modules. The default value is:

```
$(INSTALL_MOD_PATH)/lib/modules/$(KERNELRELEASE)
```

The value can be overridden in which case the default value is ignored.

3.2.32 INSTALL_MOD_PATH

INSTALL_MOD_PATH specifies a prefix to MODLIB for module directory relocations required by build roots. This is not defined in the makefile but the argument can be passed to make if needed.

3.2.33 INSTALL_MOD_STRIP

INSTALL_MOD_STRIP, if defined, will cause modules to be stripped after they are installed. If INSTALL_MOD_STRIP is '1', then the default option --strip-debug will be used. Otherwise, INSTALL_MOD_STRIP value will be used as the options to the strip command.

3.2.34 INSTALL_HDR_PATH

INSTALL_HDR_PATH specifies where to install user space headers when executing "make headers_*".

The default value is:

```
$(objtree)/usr
```

$(objtree) is the directory where output files are saved. The output directory is often set using "O=..." on the commandline.

The value can be overridden in which case the default value is ignored.

3.2.35 KBUILD_ABS_SRCTREE

Kbuild uses a relative path to point to the tree when possible. For instance, when building in the source tree, the source tree path is '.'

Setting this flag requests Kbuild to use absolute path to the source tree. There are some useful cases to do so, like when generating tag files with absolute path entries etc.

3.2.36 KBUILD_SIGN_PIN

This variable allows a passphrase or PIN to be passed to the sign-file utility when signing kernel modules, if the private key requires such.

3.2.37 KBUILD_MODPOST_WARN

KBUILD_MODPOST_WARN can be set to avoid errors in case of undefined symbols in the final module linking stage. It changes such errors into warnings.

3.2.38 KBUILD_MODPOST_NOFINAL

KBUILD_MODPOST_NOFINAL can be set to skip the final link of modules. This is solely useful to speed up test compiles.

3.2.39 KBUILD_EXTRA_SYMBOLS

For modules that use symbols from other modules. See more details in *Building External Modules*.

3.2.40 ALLSOURCE_ARCHS

For tags/TAGS/cscope targets, you can specify more than one arch to be included in the databases, separated by blank space. E.g.:

```
$ make ALLSOURCE_ARCHS="x86 mips arm" tags
```

To get all available archs you can also specify all. E.g.:

```
$ make ALLSOURCE_ARCHS=all tags
```

3.2.41 IGNORE_DIRS

For tags/TAGS/cscope targets, you can choose which directories won't be included in the databases, separated by blank space. E.g.:

```
$ make IGNORE_DIRS="drivers/gpu/drm/radeon tools" cscope
```

3.2.42 KBUILD_BUILD_TIMESTAMP

Setting this to a date string overrides the timestamp used in the UTS_VERSION definition (uname -v in the running kernel). The value has to be a string that can be passed to date -d. The default value is the output of the date command at one point during build.

3.2.43 KBUILD_BUILD_USER, KBUILD_BUILD_HOST

These two variables allow to override the user@host string displayed during boot and in /proc/version. The default value is the output of the commands whoami and host, respectively.

3.2.44 LLVM

If this variable is set to 1, Kbuild will use Clang and LLVM utilities instead of GCC and GNU binutils to build the kernel.

CHAPTER

FOUR

KCONFIG MAKE CONFIG

This file contains some assistance for using *make *config*.

Use “make help” to list all of the possible configuration targets.

The xconfig (‘qconf’), menuconfig (‘mconf’), and nconfig (‘nconf’) programs also have embedded help text. Be sure to check that for navigation, search, and other general help text.

The gconfig (‘gconf’) program has limited help text.

4.1 General

New kernel releases often introduce new config symbols. Often more important, new kernel releases may rename config symbols. When this happens, using a previously working .config file and running “make oldconfig” won’t necessarily produce a working new kernel for you, so you may find that you need to see what NEW kernel symbols have been introduced.

To see a list of new config symbols, use:

```
cp user/some/old.config .config
make listnewconfig
```

and the config program will list any new symbols, one per line.

Alternatively, you can use the brute force method:

```
make oldconfig
scripts/diffconfig .config.old .config | less
```

Environment variables for *config*

4.2 KCONFIG_CONFIG

This environment variable can be used to specify a default kernel config file name to override the default name of “.config”.

4.3 KCONFIG_DEFCONFIG_LIST

This environment variable specifies a list of config files which can be used as a base configuration in case the .config does not exist yet. Entries in the list are separated with whitespaces to each other, and the first one that exists is used.

4.4 KCONFIG_OVERWRITECONFIG

If you set KCONFIG_OVERWRITECONFIG in the environment, Kconfig will not break symlinks when .config is a symlink to somewhere else.

4.5 KCONFIG_WARN_UNKNOWN_SYMBOLS

This environment variable makes Kconfig warn about all unrecognized symbols in the config input.

4.6 KCONFIG_WERROR

If set, Kconfig treats warnings as errors.

4.7 *CONFIG_*

If you set *CONFIG_* in the environment, Kconfig will prefix all symbols with its value when saving the configuration, instead of using the default, *CONFIG_*.

Environment variables for '{allyes/allmod/allno/rand}config'

4.8 KCONFIG_ALLCONFIG

(partially based on lkml email from/by Rob Landley, re: miniconfig)

The allyesconfig/allmodconfig/allnoconfig/randconfig variants can also use the environment variable KCONFIG_ALLCONFIG as a flag or a filename that contains config symbols that the user requires to be set to a specific value. If KCONFIG_ALLCONFIG is used without a filename where KCONFIG_ALLCONFIG == "" or KCONFIG_ALLCONFIG == "1", *make* **config* checks for a file named "all{yes/mod/no/def/random}.config" (corresponding to the **config* command that was used) for symbol values that are to be forced. If this file is not found, it checks for a file named "all.config" to contain forced values.

This enables you to create "miniature" config (miniconfig) or custom config files containing just the config symbols that you are interested in. Then the kernel config system generates the full .config file, including symbols of your miniconfig file.

This 'KCONFIG_ALLCONFIG' file is a config file which contains (usually a subset of all) preset config symbols. These variable settings are still subject to normal dependency checks.

Examples:

```
KCONFIG_ALLCONFIG=custom-notebook.config make allnoconfig
```

or:

```
KCONFIG_ALLCONFIG=mini.config make allnoconfig
```

or:

```
make KCONFIG_ALLCONFIG=mini.config allnoconfig
```

These examples will disable most options (allnoconfig) but enable or disable the options that are explicitly listed in the specified mini-config files.

Environment variables for 'randconfig'

4.9 KCONFIG_SEED

You can set this to the integer value used to seed the RNG, if you want to somehow debug the behaviour of the kconfig parser/frontends. If not set, the current time will be used.

4.10 KCONFIG_PROBABILITY

This variable can be used to skew the probabilities. This variable can be unset or empty, or set to three different formats:

KCONFIG_PROBABILITY	y:n split	y:m:n split
unset or empty	50 : 50	33 : 33 : 34
N	N : 100-N	N/2 : N/2 : 100-N
[1] N:M	N+M : 100-(N+M)	N : M : 100-(N+M)
[2] N:M:L	N : 100-N	M : L : 100-(M+L)

where N, M and L are integers (in base 10) in the range [0,100], and so that:

[1] N+M is in the range [0,100]

[2] M+L is in the range [0,100]

Examples:

```
KCONFIG_PROBABILITY=10
        10% of booleans will be set to 'y', 90% to 'n'
        5% of tristates will be set to 'y', 5% to 'm', 90% to 'n'
KCONFIG_PROBABILITY=15:25
        40% of booleans will be set to 'y', 60% to 'n'
```

```
        15% of tristates will be set to 'y', 25% to 'm', 60% to 'n'
KCONFIG_PROBABILITY=10:15:15
        10% of booleans will be set to 'y', 90% to 'n'
        15% of tristates will be set to 'y', 15% to 'm', 70% to 'n'
```

Environment variables for ‘syncconfig’

4.11 KCONFIG_NOSILENTUPDATE

If this variable has a non-blank value, it prevents silent kernel config updates (requires explicit updates).

4.12 KCONFIG_AUTOCONFIG

This environment variable can be set to specify the path & name of the “auto.conf” file. Its default value is “include/config/auto.conf”.

4.13 KCONFIG_AUTOHEADER

This environment variable can be set to specify the path & name of the “autoconf.h” (header) file. Its default value is “include/generated/autoconf.h”.

4.14 menuconfig

SEARCHING for CONFIG symbols

Searching in menuconfig:

> The Search function searches for kernel configuration symbol names, so you have to know something close to what you are looking for.
>
> Example:
>
> ```
> /hotplug
> This lists all config symbols that contain "hotplug",
> e.g., HOTPLUG_CPU, MEMORY_HOTPLUG.
> ```
>
> For search help, enter / followed by TAB-TAB (to highlight <Help>) and Enter. This will tell you that you can also use regular expressions (regexes) in the search string, so if you are not interested in MEMORY_HOTPLUG, you could try:
>
> ```
> /^hotplug
> ```
>
> When searching, symbols are sorted thus:

- first, exact matches, sorted alphabetically (an exact match is when the search matches the complete symbol name);
- then, other matches, sorted alphabetically.

For example: ^ATH.K matches:

> ATH5K ATH9K ATH5K_AHB ATH5K_DEBUG [...] ATH6KL ATH6KL_DEBUG [...] ATH9K_AHB ATH9K_BTCOEX_SUPPORT ATH9K_COMMON [...]

of which only ATH5K and ATH9K match exactly and so are sorted first (and in alphabetical order), then come all other symbols, sorted in alphabetical order.

In this menu, pressing the key in the (#) prefix will jump directly to that location. You will be returned to the current search results after exiting this new menu.

User interface options for 'menuconfig'

4.15 MENUCONFIG_COLOR

It is possible to select different color themes using the variable MENUCONFIG_COLOR. To select a theme use:

```
make MENUCONFIG_COLOR=<theme> menuconfig
```

Available themes are:

```
- mono       => selects colors suitable for monochrome displays
- blackbg    => selects a color scheme with black background
- classic    => theme with blue background. The classic look
- bluetitle  => a LCD friendly version of classic. (default)
```

4.16 MENUCONFIG_MODE

This mode shows all sub-menus in one large tree.

Example:

```
make MENUCONFIG_MODE=single_menu menuconfig
```

4.17 nconfig

nconfig is an alternate text-based configurator. It lists function keys across the bottom of the terminal (window) that execute commands. You can also just use the corresponding numeric key to execute the commands unless you are in a data entry window. E.g., instead of F6 for Save, you can just press 6.

Use F1 for Global help or F3 for the Short help menu.

Searching in nconfig:

> You can search either in the menu entry "prompt" strings or in the configuration symbols.
>
> Use / to begin a search through the menu entries. This does not support regular expressions. Use <Down> or <Up> for Next hit and Previous hit, respectively. Use <Esc> to terminate the search mode.
>
> F8 (SymSearch) searches the configuration symbols for the given string or regular expression (regex).
>
> In the SymSearch, pressing the key in the (#) prefix will jump directly to that location. You will be returned to the current search results after exiting this new menu.

4.18 NCONFIG_MODE

This mode shows all sub-menus in one large tree.

Example:

```
make NCONFIG_MODE=single_menu nconfig
```

4.19 xconfig

Searching in xconfig:

> The Search function searches for kernel configuration symbol names, so you have to know something close to what you are looking for.
>
> Example:
>
> ```
> Ctrl-F hotplug
> ```
>
> or:
>
> ```
> Menu: File, Search, hotplug
> ```
>
> lists all config symbol entries that contain "hotplug" in the symbol name. In this Search dialog, you may change the config setting for any of the entries that are not grayed out. You can also enter a different search string without having to return to the main menu.

4.20 gconfig

Searching in gconfig:

> There is no search command in gconfig. However, gconfig does have several different viewing choices, modes, and options.

CHAPTER

FIVE

LINUX KERNEL MAKEFILES

This document describes the Linux kernel Makefiles.

5.1 Overview

The Makefiles have five parts:

```
Makefile                    the top Makefile.
.config                     the kernel configuration file.
arch/$(SRCARCH)/Makefile    the arch Makefile.
scripts/Makefile.*          common rules etc. for all kbuild Makefiles.
kbuild Makefiles            exist in every subdirectory
```

The top Makefile reads the .config file, which comes from the kernel configuration process.

The top Makefile is responsible for building two major products: vmlinux (the resident kernel image) and modules (any module files). It builds these goals by recursively descending into the subdirectories of the kernel source tree.

The list of subdirectories which are visited depends upon the kernel configuration. The top Makefile textually includes an arch Makefile with the name arch/$(SRCARCH)/Makefile. The arch Makefile supplies architecture-specific information to the top Makefile.

Each subdirectory has a kbuild Makefile which carries out the commands passed down from above. The kbuild Makefile uses information from the .config file to construct various file lists used by kbuild to build any built-in or modular targets.

scripts/Makefile.* contains all the definitions/rules etc. that are used to build the kernel based on the kbuild makefiles.

5.2 Who does what

People have four different relationships with the kernel Makefiles.

Users are people who build kernels. These people type commands such as `make menuconfig` or `make`. They usually do not read or edit any kernel Makefiles (or any other source files).

Normal developers are people who work on features such as device drivers, file systems, and network protocols. These people need to maintain the kbuild Makefiles for the subsystem they are working on. In order to do this effectively, they need some overall knowledge about the kernel Makefiles, plus detailed knowledge about the public interface for kbuild.

Arch developers are people who work on an entire architecture, such as sparc or ia64. Arch developers need to know about the arch Makefile as well as kbuild Makefiles.

Kbuild developers are people who work on the kernel build system itself. These people need to know about all aspects of the kernel Makefiles.

This document is aimed towards normal developers and arch developers.

5.3 The kbuild files

Most Makefiles within the kernel are kbuild Makefiles that use the kbuild infrastructure. This chapter introduces the syntax used in the kbuild makefiles.

The preferred name for the kbuild files are `Makefile` but `Kbuild` can be used and if both a `Makefile` and a `Kbuild` file exists, then the `Kbuild` file will be used.

Section *Goal definitions* is a quick intro; further chapters provide more details, with real examples.

5.3.1 Goal definitions

Goal definitions are the main part (heart) of the kbuild Makefile. These lines define the files to be built, any special compilation options, and any subdirectories to be entered recursively.

The most simple kbuild makefile contains one line:

Example:

```
obj-y += foo.o
```

This tells kbuild that there is one object in that directory, named foo.o. foo.o will be built from foo.c or foo.S.

If foo.o shall be built as a module, the variable obj-m is used. Therefore the following pattern is often used:

Example:

```
obj-$(CONFIG_FOO) += foo.o
```

$(CONFIG_FOO) evaluates to either y (for built-in) or m (for module). If CONFIG_FOO is neither y nor m, then the file will not be compiled nor linked.

5.3.2 Built-in object goals - obj-y

The kbuild Makefile specifies object files for vmlinux in the $(obj-y) lists. These lists depend on the kernel configuration.

Kbuild compiles all the $(obj-y) files. It then calls `$(AR) rcSTP` to merge these files into one built-in.a file. This is a thin archive without a symbol table. It will be later linked into vmlinux by scripts/link-vmlinux.sh

The order of files in $(obj-y) is significant. Duplicates in the lists are allowed: the first instance will be linked into built-in.a and succeeding instances will be ignored.

Link order is significant, because certain functions (module_init() / __initcall) will be called during boot in the order they appear. So keep in mind that changing the link order may e.g. change the order in which your SCSI controllers are detected, and thus your disks are renumbered.

Example:

```
#drivers/isdn/i4l/Makefile
# Makefile for the kernel ISDN subsystem and device drivers.
# Each configuration option enables a list of files.
obj-$(CONFIG_ISDN_I4L)         += isdn.o
obj-$(CONFIG_ISDN_PPP_BSDCOMP) += isdn_bsdcomp.o
```

5.3.3 Loadable module goals - obj-m

$(obj-m) specifies object files which are built as loadable kernel modules.

A module may be built from one source file or several source files. In the case of one source file, the kbuild makefile simply adds the file to $(obj-m).

Example:

```
#drivers/isdn/i4l/Makefile
obj-$(CONFIG_ISDN_PPP_BSDCOMP) += isdn_bsdcomp.o
```

Note: In this example $(CONFIG_ISDN_PPP_BSDCOMP) evaluates to “m”

If a kernel module is built from several source files, you specify that you want to build a module in the same way as above; however, kbuild needs to know which object files you want to build your module from, so you have to tell it by setting a $(<module_name>-y) variable.

Example:

```
#drivers/isdn/i4l/Makefile
obj-$(CONFIG_ISDN_I4L) += isdn.o
isdn-y := isdn_net_lib.o isdn_v110.o isdn_common.o
```

In this example, the module name will be isdn.o. Kbuild will compile the objects listed in $(isdn-y) and then run `$(LD) -r` on the list of these files to generate isdn.o.

Due to kbuild recognizing $(<module_name>-y) for composite objects, you can use the value of a `CONFIG_` symbol to optionally include an object file as part of a composite object.

Example:

```
#fs/ext2/Makefile
obj-$(CONFIG_EXT2_FS) += ext2.o
ext2-y := balloc.o dir.o file.o ialloc.o inode.o ioctl.o \
  namei.o super.o symlink.o
ext2-$(CONFIG_EXT2_FS_XATTR) += xattr.o xattr_user.o \
  xattr_trusted.o
```

In this example, xattr.o, xattr_user.o and xattr_trusted.o are only part of the composite object ext2.o if $(CONFIG_EXT2_FS_XATTR) evaluates to “y”.

Note: Of course, when you are building objects into the kernel, the syntax above will also work. So, if you have CONFIG_EXT2_FS=y, kbuild will build an ext2.o file for you out of the individual parts and then link this into built-in.a, as you would expect.

5.3.4 Library file goals - lib-y

Objects listed with obj-* are used for modules, or combined in a built-in.a for that specific directory. There is also the possibility to list objects that will be included in a library, lib.a. All objects listed with lib-y are combined in a single library for that directory. Objects that are listed in obj-y and additionally listed in lib-y will not be included in the library, since they will be accessible anyway. For consistency, objects listed in lib-m will be included in lib.a.

Note that the same kbuild makefile may list files to be built-in and to be part of a library. Therefore the same directory may contain both a built-in.a and a lib.a file.

Example:

```
#arch/x86/lib/Makefile
lib-y    := delay.o
```

This will create a library lib.a based on delay.o. For kbuild to actually recognize that there is a lib.a being built, the directory shall be listed in libs-y.

See also *List directories to visit when descending*.

Use of lib-y is normally restricted to `lib/` and `arch/*/lib`.

5.3.5 Descending down in directories

A Makefile is only responsible for building objects in its own directory. Files in subdirectories should be taken care of by Makefiles in these subdirs. The build system will automatically invoke make recursively in subdirectories, provided you let it know of them.

To do so, obj-y and obj-m are used. ext2 lives in a separate directory, and the Makefile present in fs/ tells kbuild to descend down using the following assignment.

Example:

```
#fs/Makefile
obj-$(CONFIG_EXT2_FS) += ext2/
```

If CONFIG_EXT2_FS is set to either “y” (built-in) or “m” (modular) the corresponding obj- variable will be set, and kbuild will descend down in the ext2 directory.

Kbuild uses this information not only to decide that it needs to visit the directory, but also to decide whether or not to link objects from the directory into vmlinux.

When Kbuild descends into the directory with “y”, all built-in objects from that directory are combined into the built-in.a, which will be eventually linked into vmlinux.

When Kbuild descends into the directory with “m”, in contrast, nothing from that directory will be linked into vmlinux. If the Makefile in that directory specifies obj-y, those objects will be left orphan. It is very likely a bug of the Makefile or of dependencies in Kconfig.

Kbuild also supports dedicated syntax, subdir-y and subdir-m, for descending into subdirectories. It is a good fit when you know they do not contain kernel-space objects at all. A typical usage is to let Kbuild descend into subdirectories to build tools.

Examples:

```
# scripts/Makefile
subdir-$(CONFIG_GCC_PLUGINS) += gcc-plugins
subdir-$(CONFIG_MODVERSIONS) += genksyms
subdir-$(CONFIG_SECURITY_SELINUX) += selinux
```

Unlike obj-y/m, subdir-y/m does not need the trailing slash since this syntax is always used for directories.

It is good practice to use a `CONFIG_` variable when assigning directory names. This allows kbuild to totally skip the directory if the corresponding `CONFIG_` option is neither "y" nor "m".

5.3.6 Non-builtin vmlinux targets - extra-y

extra-y specifies targets which are needed for building vmlinux, but not combined into built-in.a.

Examples are:

1) vmlinux linker script

   The linker script for vmlinux is located at arch/$(SRCARCH)/kernel/vmlinux.lds

Example:

```
# arch/x86/kernel/Makefile
extra-y       += vmlinux.lds
```

$(extra-y) should only contain targets needed for vmlinux.

Kbuild skips extra-y when vmlinux is apparently not a final goal. (e.g. `make modules`, or building external modules)

If you intend to build targets unconditionally, always-y (explained in the next section) is the correct syntax to use.

5.3.7 Always built goals - always-y

always-y specifies targets which are literally always built when Kbuild visits the Makefile.

Example:

```
# ./Kbuild
offsets-file := include/generated/asm-offsets.h
always-y += $(offsets-file)
```

5.3.8 Compilation flags

ccflags-y, asflags-y and ldflags-y
These three flags apply only to the kbuild makefile in which they are assigned. They are used for all the normal cc, as and ld invocations happening during a recursive build. Note: Flags with the same behaviour were previously named: EXTRA_CFLAGS, EXTRA_AFLAGS and EXTRA_LDFLAGS. They are still supported but their usage is deprecated.

ccflags-y specifies options for compiling with $(CC).

Example:

```
# drivers/acpi/acpica/Makefile
ccflags-y                               := -Os -D_LINUX -DBUILDING_ACPICA
ccflags-$(CONFIG_ACPI_DEBUG)            += -DACPI_DEBUG_OUTPUT
```

This variable is necessary because the top Makefile owns the variable $(KBUILD_CFLAGS) and uses it for compilation flags for the entire tree.

asflags-y specifies assembler options.

Example:

```
#arch/sparc/kernel/Makefile
asflags-y := -ansi
```

ldflags-y specifies options for linking with $(LD).

Example:

```
#arch/cris/boot/compressed/Makefile
ldflags-y += -T $(srctree)/$(src)/decompress_$(arch-y).lds
```

subdir-ccflags-y, subdir-asflags-y
The two flags listed above are similar to ccflags-y and asflags-y. The difference is that the subdir- variants have effect for the kbuild file where they are present and all subdirectories. Options specified using subdir-* are added to the commandline before the options specified using the non-subdir variants.

Example:

```
subdir-ccflags-y := -Werror
```

ccflags-remove-y, asflags-remove-y
These flags are used to remove particular flags for the compiler, assembler invocations.

Example:

```
ccflags-remove-$(CONFIG_MCOUNT) += -pg
```

CFLAGS_$@, AFLAGS_$@
CFLAGS_$@ and AFLAGS_$@ only apply to commands in current kbuild makefile.

$(CFLAGS_$@) specifies per-file options for $(CC). The $@ part has a literal value which specifies the file that it is for.

CFLAGS_$@ has the higher priority than ccflags-remove-y; CFLAGS_$@ can re-add compiler flags that were removed by ccflags-remove-y.

Example:

```
# drivers/scsi/Makefile
CFLAGS_aha152x.o =   -DAHA152X_STAT -DAUTOCONF
```

This line specify compilation flags for aha152x.o.

$(AFLAGS_$@) is a similar feature for source files in assembly languages.

AFLAGS_$@ has the higher priority than asflags-remove-y; AFLAGS_$@ can re-add assembler flags that were removed by asflags-remove-y.

Example:

```
# arch/arm/kernel/Makefile
AFLAGS_head.o        := -DTEXT_OFFSET=$(TEXT_OFFSET)
AFLAGS_crunch-bits.o := -Wa,-mcpu=ep9312
AFLAGS_iwmmxt.o      := -Wa,-mcpu=iwmmxt
```

5.3.9 Dependency tracking

Kbuild tracks dependencies on the following:

1) All prerequisite files (both `*.c` and `*.h`)
2) `CONFIG_` options used in all prerequisite files
3) Command-line used to compile target

Thus, if you change an option to $(CC) all affected files will be re-compiled.

5.3.10 Custom Rules

Custom rules are used when the kbuild infrastructure does not provide the required support. A typical example is header files generated during the build process. Another example are the architecture-specific Makefiles which need custom rules to prepare boot images etc.

Custom rules are written as normal Make rules. Kbuild is not executing in the directory where the Makefile is located, so all custom rules shall use a relative path to prerequisite files and target files.

Two variables are used when defining custom rules:

$(src)
$(src) is a relative path which points to the directory where the Makefile is located. Always use $(src) when referring to files located in the src tree.

$(obj)
$(obj) is a relative path which points to the directory where the target is saved. Always use $(obj) when referring to generated files.

Example:

```
#drivers/scsi/Makefile
$(obj)/53c8xx_d.h: $(src)/53c7,8xx.scr $(src)/script_asm.pl
$(CPP) -DCHIP=810 - < $< | ... $(src)/script_asm.pl
```

This is a custom rule, following the normal syntax required by make.

The target file depends on two prerequisite files. References to the target file are prefixed with $(obj), references to prerequisites are referenced with $(src) (because they are not generated files).

$(kecho)

echoing information to user in a rule is often a good practice but when execution `make -s` one does not expect to see any output except for warnings/errors. To support this kbuild defines $(kecho) which will echo out the text following $(kecho) to stdout except if `make -s` is used.

Example:

```
# arch/arm/Makefile
$(BOOT_TARGETS): vmlinux
        $(Q)$(MAKE) $(build)=$(boot) MACHINE=$(MACHINE) $(boot)/$@
        @$(kecho) '  Kernel: $(boot)/$@ is ready'
```

When kbuild is executing with KBUILD_VERBOSE unset, then only a shorthand of a command is normally displayed. To enable this behaviour for custom commands kbuild requires two variables to be set:

```
quiet_cmd_<command> - what shall be echoed
      cmd_<command> - the command to execute
```

Example:

```
# lib/Makefile
quiet_cmd_crc32 = GEN     $@
      cmd_crc32 = $< > $@

$(obj)/crc32table.h: $(obj)/gen_crc32table
        $(call cmd,crc32)
```

When updating the $(obj)/crc32table.h target, the line:

```
GEN     lib/crc32table.h
```

will be displayed with `make KBUILD_VERBOSE=`.

5.3.11 Command change detection

When the rule is evaluated, timestamps are compared between the target and its prerequisite files. GNU Make updates the target when any of the prerequisites is newer than that.

The target should be rebuilt also when the command line has changed since the last invocation. This is not supported by Make itself, so Kbuild achieves this by a kind of meta-programming.

if_changed is the macro used for this purpose, in the following form:

```
quiet_cmd_<command> = ...
      cmd_<command> = ...

<target>: <source(s)> FORCE
        $(call if_changed,<command>)
```

Any target that utilizes if_changed must be listed in $(targets), otherwise the command line check will fail, and the target will always be built.

If the target is already listed in the recognized syntax such as obj-y/m, lib-y/m, extra-y/m, always-y/m, hostprogs, userprogs, Kbuild automatically adds it to $(targets). Otherwise, the target must be explicitly added to $(targets).

Assignments to $(targets) are without $(obj)/ prefix. if_changed may be used in conjunction with custom rules as defined in *Custom Rules*.

Note: It is a typical mistake to forget the FORCE prerequisite. Another common pitfall is that whitespace is sometimes significant; for instance, the below will fail (note the extra space after the comma):

```
target: source(s) FORCE
```

WRONG! $(call if_changed, objcopy)

Note:
if_changed should not be used more than once per target. It stores the executed command in a corresponding .cmd file and multiple calls would result in overwrites and unwanted results when the target is up to date and only the tests on changed commands trigger execution of commands.

5.3.12 $(CC) support functions

The kernel may be built with several different versions of $(CC), each supporting a unique set of features and options. kbuild provides basic support to check for valid options for $(CC). $(CC) is usually the gcc compiler, but other alternatives are available.

as-option
as-option is used to check if $(CC) -- when used to compile assembler (`*.S`) files -- supports the given option. An optional second option may be specified if the first option is not supported.

Example:

```
#arch/sh/Makefile
cflags-y += $(call as-option,-Wa$(comma)-isa=$(isa-y),)
```

In the above example, cflags-y will be assigned the option -Wa$(comma)-isa=$(isa-y) if it is supported by $(CC). The second argument is optional, and if supplied will be used if first argument is not supported.

as-instr
as-instr checks if the assembler reports a specific instruction and then outputs either option1 or option2 C escapes are supported in the test instruction Note: as-instr-option uses KBUILD_AFLAGS for assembler options

cc-option
cc-option is used to check if $(CC) supports a given option, and if not supported to use an optional second option.

Example:

```
#arch/x86/Makefile
cflags-y += $(call cc-option,-march=pentium-mmx,-march=i586)
```

In the above example, cflags-y will be assigned the option -march=pentium-mmx if supported by $(CC), otherwise -march=i586. The second argument to cc-option is optional, and if omitted, cflags-y will be assigned no value if first option is not supported. Note: cc-option uses KBUILD_CFLAGS for $(CC) options

cc-option-yn
cc-option-yn is used to check if gcc supports a given option and return "y" if supported, otherwise "n".

Example:

```
#arch/ppc/Makefile
biarch := $(call cc-option-yn, -m32)
aflags-$(biarch) += -a32
cflags-$(biarch) += -m32
```

In the above example, $(biarch) is set to y if $(CC) supports the -m32 option. When $(biarch) equals "y", the expanded variables $(aflags-y) and $(cflags-y) will be assigned the values -a32 and -m32, respectively.

Note: cc-option-yn uses KBUILD_CFLAGS for $(CC) options

cc-disable-warning
cc-disable-warning checks if gcc supports a given warning and returns the commandline switch to disable it. This special function is needed, because gcc 4.4 and later accept any unknown -Wno-* option and only warn about it if there is another warning in the source file.

Example:

```
KBUILD_CFLAGS += $(call cc-disable-warning, unused-but-set-variable)
```

In the above example, -Wno-unused-but-set-variable will be added to KBUILD_CFLAGS only if gcc really accepts it.

gcc-min-version
gcc-min-version tests if the value of $(CONFIG_GCC_VERSION) is greater than or equal to the provided value and evaluates to y if so.

Example:

```
cflags-$(call gcc-min-version, 70100) := -foo
```

In this example, cflags-y will be assigned the value -foo if $(CC) is gcc and $(CONFIG_GCC_VERSION) is >= 7.1.

clang-min-version
clang-min-version tests if the value of $(CONFIG_CLANG_VERSION) is greater than or equal to the provided value and evaluates to y if so.

Example:

```
cflags-$(call clang-min-version, 110000) := -foo
```

In this example, cflags-y will be assigned the value -foo if $(CC) is clang and $(CONFIG_CLANG_VERSION) is >= 11.0.0.

cc-cross-prefix
cc-cross-prefix is used to check if there exists a $(CC) in path with one of the listed prefixes. The first prefix where there exist a prefix$(CC) in the PATH is returned - and if no prefix$(CC) is found then nothing is returned.

Additional prefixes are separated by a single space in the call of cc-cross-prefix.

This functionality is useful for architecture Makefiles that try to set CROSS_COMPILE to well-known values but may have several values to select between.

It is recommended only to try to set CROSS_COMPILE if it is a cross build (host arch is different from target arch). And if CROSS_COMPILE is already set then leave it with the old value.

Example:

```
#arch/m68k/Makefile
ifneq ($(SUBARCH),$(ARCH))
        ifeq ($(CROSS_COMPILE),)
                CROSS_COMPILE := $(call cc-cross-prefix, m68k-linux-gnu-)
        endif
endif
```

5.3.13 $(LD) support functions

ld-option
ld-option is used to check if $(LD) supports the supplied option. ld-option takes two options as arguments.

The second argument is an optional option that can be used if the first option is not supported by $(LD).

Example:

```
#Makefile
LDFLAGS_vmlinux += $(call ld-option, -X)
```

5.3.14 Script invocation

Make rules may invoke scripts to build the kernel. The rules shall always provide the appropriate interpreter to execute the script. They shall not rely on the execute bits being set, and shall not invoke the script directly. For the convenience of manual script invocation, such as invoking ./scripts/checkpatch.pl, it is recommended to set execute bits on the scripts nonetheless.

Kbuild provides variables $(CONFIG_SHELL), $(AWK), $(PERL), and $(PYTHON3) to refer to interpreters for the respective scripts.

Example:

```
#Makefile
cmd_depmod = $(CONFIG_SHELL) $(srctree)/scripts/depmod.sh $(DEPMOD) \
        $(KERNELRELEASE)
```

5.4 Host Program support

Kbuild supports building executables on the host for use during the compilation stage.

Two steps are required in order to use a host executable.

The first step is to tell kbuild that a host program exists. This is done utilising the variable `hostprogs`.

The second step is to add an explicit dependency to the executable. This can be done in two ways. Either add the dependency in a rule, or utilise the variable `always-y`. Both possibilities are described in the following.

5.4.1 Simple Host Program

In some cases there is a need to compile and run a program on the computer where the build is running.

The following line tells kbuild that the program bin2hex shall be built on the build host.

Example:

```
hostprogs := bin2hex
```

Kbuild assumes in the above example that bin2hex is made from a single c-source file named bin2hex.c located in the same directory as the Makefile.

5.4.2 Composite Host Programs

Host programs can be made up based on composite objects. The syntax used to define composite objects for host programs is similar to the syntax used for kernel objects. $(<executable>-objs) lists all objects used to link the final executable.

Example:

```
#scripts/lxdialog/Makefile
hostprogs     := lxdialog
lxdialog-objs := checklist.o lxdialog.o
```

Objects with extension .o are compiled from the corresponding .c files. In the above example, checklist.c is compiled to checklist.o and lxdialog.c is compiled to lxdialog.o.

Finally, the two .o files are linked to the executable, lxdialog. Note: The syntax <executable>-y is not permitted for host-programs.

5.4.3 Using C++ for host programs

kbuild offers support for host programs written in C++. This was introduced solely to support kconfig, and is not recommended for general use.

Example:

```
#scripts/kconfig/Makefile
hostprogs     := qconf
qconf-cxxobjs := qconf.o
```

In the example above the executable is composed of the C++ file qconf.cc - identified by $(qconf-cxxobjs).

If qconf is composed of a mixture of .c and .cc files, then an additional line can be used to identify this.

Example:

```
#scripts/kconfig/Makefile
hostprogs     := qconf
qconf-cxxobjs := qconf.o
qconf-objs    := check.o
```

5.4.4 Using Rust for host programs

Kbuild offers support for host programs written in Rust. However, since a Rust toolchain is not mandatory for kernel compilation, it may only be used in scenarios where Rust is required to be available (e.g. when `CONFIG_RUST` is enabled).

Example:

```
hostprogs     := target
target-rust   := y
```

Kbuild will compile `target` using `target.rs` as the crate root, located in the same directory as the `Makefile`. The crate may consist of several source files (see `samples/rust/hostprogs`).

5.4.5 Controlling compiler options for host programs

When compiling host programs, it is possible to set specific flags. The programs will always be compiled utilising $(HOSTCC) passed the options specified in $(KBUILD_HOSTCFLAGS).

To set flags that will take effect for all host programs created in that Makefile, use the variable HOST_EXTRACFLAGS.

Example:

```
#scripts/lxdialog/Makefile
HOST_EXTRACFLAGS += -I/usr/include/ncurses
```

To set specific flags for a single file the following construction is used:

Example:

```
#arch/ppc64/boot/Makefile
HOSTCFLAGS_piggyback.o := -DKERNELBASE=$(KERNELBASE)
```

It is also possible to specify additional options to the linker.

Example:

```
#scripts/kconfig/Makefile
HOSTLDLIBS_qconf := -L$(QTDIR)/lib
```

When linking qconf, it will be passed the extra option `-L$(QTDIR)/lib`.

5.4.6 When host programs are actually built

Kbuild will only build host-programs when they are referenced as a prerequisite.

This is possible in two ways:

(1) List the prerequisite explicitly in a custom rule.

Example:

```
#drivers/pci/Makefile
hostprogs := gen-devlist
$(obj)/devlist.h: $(src)/pci.ids $(obj)/gen-devlist
( cd $(obj); ./gen-devlist ) < $<
```

The target $(obj)/devlist.h will not be built before $(obj)/gen-devlist is updated. Note that references to the host programs in custom rules must be prefixed with $(obj).

(2) Use always-y

When there is no suitable custom rule, and the host program shall be built when a makefile is entered, the always-y variable shall be used.

Example:

```
#scripts/lxdialog/Makefile
hostprogs     := lxdialog
always-y      := $(hostprogs)
```

Kbuild provides the following shorthand for this:

```
hostprogs-always-y := lxdialog
```

This will tell kbuild to build lxdialog even if not referenced in any rule.

5.5 Userspace Program support

Just like host programs, Kbuild also supports building userspace executables for the target architecture (i.e. the same architecture as you are building the kernel for).

The syntax is quite similar. The difference is to use `userprogs` instead of `hostprogs`.

5.5.1 Simple Userspace Program

The following line tells kbuild that the program bpf-direct shall be built for the target architecture.

Example:

```
userprogs := bpf-direct
```

Kbuild assumes in the above example that bpf-direct is made from a single C source file named bpf-direct.c located in the same directory as the Makefile.

5.5.2 Composite Userspace Programs

Userspace programs can be made up based on composite objects. The syntax used to define composite objects for userspace programs is similar to the syntax used for kernel objects. $(<executable>-objs) lists all objects used to link the final executable.

Example:

```
#samples/seccomp/Makefile
userprogs      := bpf-fancy
bpf-fancy-objs := bpf-fancy.o bpf-helper.o
```

Objects with extension .o are compiled from the corresponding .c files. In the above example, bpf-fancy.c is compiled to bpf-fancy.o and bpf-helper.c is compiled to bpf-helper.o.

Finally, the two .o files are linked to the executable, bpf-fancy. Note: The syntax <executable>-y is not permitted for userspace programs.

5.5.3 Controlling compiler options for userspace programs

When compiling userspace programs, it is possible to set specific flags. The programs will always be compiled utilising $(CC) passed the options specified in $(KBUILD_USERCFLAGS).

To set flags that will take effect for all userspace programs created in that Makefile, use the variable userccflags.

Example:

```
# samples/seccomp/Makefile
userccflags += -I usr/include
```

To set specific flags for a single file the following construction is used:

Example:

```
bpf-helper-userccflags += -I user/include
```

It is also possible to specify additional options to the linker.

Example:

```
# net/bpfilter/Makefile
bpfilter_umh-userldflags += -static
```

When linking bpfilter_umh, it will be passed the extra option -static.

From command line, *USERCFLAGS and USERLDFLAGS* will also be used.

5.5.4 When userspace programs are actually built

Kbuild builds userspace programs only when told to do so. There are two ways to do this.

(1) Add it as the prerequisite of another file

Example:

```
#net/bpfilter/Makefile
userprogs := bpfilter_umh
$(obj)/bpfilter_umh_blob.o: $(obj)/bpfilter_umh
```

$(obj)/bpfilter_umh is built before $(obj)/bpfilter_umh_blob.o

(2) Use always-y

Example:

```
userprogs := binderfs_example
always-y := $(userprogs)
```

Kbuild provides the following shorthand for this:

```
userprogs-always-y := binderfs_example
```

This will tell Kbuild to build binderfs_example when it visits this Makefile.

5.6 Kbuild clean infrastructure

`make clean` deletes most generated files in the obj tree where the kernel is compiled. This includes generated files such as host programs. Kbuild knows targets listed in $(hostprogs), $(always-y), $(always-m), $(always-), $(extra-y), $(extra-) and $(targets). They are all deleted during `make clean`. Files matching the patterns `*.[oas]`, `*.ko`, plus some additional files generated by kbuild are deleted all over the kernel source tree when `make clean` is executed.

Additional files or directories can be specified in kbuild makefiles by use of $(clean-files).

Example:

```
#lib/Makefile
clean-files := crc32table.h
```

When executing `make clean`, the file `crc32table.h` will be deleted. Kbuild will assume files to be in the same relative directory as the Makefile.

To exclude certain files or directories from make clean, use the $(no-clean-files) variable.

Usually kbuild descends down in subdirectories due to `obj-* := dir/`, but in the architecture makefiles where the kbuild infrastructure is not sufficient this sometimes needs to be explicit.

Example:

```
#arch/x86/boot/Makefile
subdir- := compressed
```

The above assignment instructs kbuild to descend down in the directory compressed/ when `make clean` is executed.

Note 1: arch/$(SRCARCH)/Makefile cannot use `subdir-`, because that file is included in the top level makefile. Instead, arch/$(SRCARCH)/Kbuild can use `subdir-`.

Note 2: All directories listed in core-y, libs-y, drivers-y and net-y will be visited during `make clean`.

5.7 Architecture Makefiles

The top level Makefile sets up the environment and does the preparation, before starting to descend down in the individual directories.

The top level makefile contains the generic part, whereas arch/$(SRCARCH)/Makefile contains what is required to set up kbuild for said architecture.

To do so, arch/$(SRCARCH)/Makefile sets up a number of variables and defines a few targets.

When kbuild executes, the following steps are followed (roughly):

1) Configuration of the kernel => produce .config
2) Store kernel version in include/linux/version.h
3) Updating all other prerequisites to the target prepare:
   - Additional prerequisites are specified in arch/$(SRCARCH)/Makefile

4) Recursively descend down in all directories listed in init-* core* drivers-* net-* libs-* and build all targets.
   - The values of the above variables are expanded in arch/$(SRCARCH)/Makefile.
5) All object files are then linked and the resulting file vmlinux is located at the root of the obj tree. The very first objects linked are listed in scripts/head-object-list.txt.
6) Finally, the architecture-specific part does any required post processing and builds the final bootimage.
   - This includes building boot records
   - Preparing initrd images and the like

5.7.1 Set variables to tweak the build to the architecture

KBUILD_LDFLAGS
Generic $(LD) options

Flags used for all invocations of the linker. Often specifying the emulation is sufficient.

Example:

```
#arch/s390/Makefile
KBUILD_LDFLAGS         := -m elf_s390
```

Note: ldflags-y can be used to further customise the flags used. See *Non-builtin vmlinux targets - extra-y*.

LDFLAGS_vmlinux
Options for $(LD) when linking vmlinux

LDFLAGS_vmlinux is used to specify additional flags to pass to the linker when linking the final vmlinux image.

LDFLAGS_vmlinux uses the LDFLAGS_$@ support.

Example:

```
#arch/x86/Makefile
LDFLAGS_vmlinux := -e stext
```

OBJCOPYFLAGS
objcopy flags

When $(call if_changed,objcopy) is used to translate a .o file, the flags specified in OBJCOPYFLAGS will be used.

$(call if_changed,objcopy) is often used to generate raw binaries on vmlinux.

Example:

```
#arch/s390/Makefile
OBJCOPYFLAGS := -O binary

#arch/s390/boot/Makefile
```

```
$(obj)/image: vmlinux FORCE
        $(call if_changed,objcopy)
```

In this example, the binary $(obj)/image is a binary version of vmlinux. The usage of $(call if_changed,xxx) will be described later.

KBUILD_AFLAGS
Assembler flags

Default value - see top level Makefile.

Append or modify as required per architecture.

Example:

```
#arch/sparc64/Makefile
KBUILD_AFLAGS += -m64 -mcpu=ultrasparc
```

KBUILD_CFLAGS
$(CC) compiler flags

Default value - see top level Makefile.

Append or modify as required per architecture.

Often, the KBUILD_CFLAGS variable depends on the configuration.

Example:

```
#arch/x86/boot/compressed/Makefile
cflags-$(CONFIG_X86_32) := -march=i386
cflags-$(CONFIG_X86_64) := -mcmodel=small
KBUILD_CFLAGS += $(cflags-y)
```

Many arch Makefiles dynamically run the target C compiler to probe supported options:

```
#arch/x86/Makefile

...
cflags-$(CONFIG_MPENTIUMII)     += $(call cc-option,\
                                        -march=pentium2,-march=i686)
...
# Disable unit-at-a-time mode ...
KBUILD_CFLAGS += $(call cc-option,-fno-unit-at-a-time)
...
```

The first example utilises the trick that a config option expands to “y” when selected.

KBUILD_RUSTFLAGS
$(RUSTC) compiler flags

Default value - see top level Makefile.

Append or modify as required per architecture.

Often, the KBUILD_RUSTFLAGS variable depends on the configuration.

Note that target specification file generation (for `--target`) is handled in `scripts/generate_rust_target.rs`.

KBUILD_AFLAGS_KERNEL
Assembler options specific for built-in

$(KBUILD_AFLAGS_KERNEL) contains extra C compiler flags used to compile resident kernel code.

KBUILD_AFLAGS_MODULE
Assembler options specific for modules

$(KBUILD_AFLAGS_MODULE) is used to add arch-specific options that are used for assembler.

From commandline AFLAGS_MODULE shall be used (see *Kbuild*).

KBUILD_CFLAGS_KERNEL
$(CC) options specific for built-in

$(KBUILD_CFLAGS_KERNEL) contains extra C compiler flags used to compile resident kernel code.

KBUILD_CFLAGS_MODULE
Options for $(CC) when building modules

$(KBUILD_CFLAGS_MODULE) is used to add arch-specific options that are used for $(CC).

From commandline CFLAGS_MODULE shall be used (see *Kbuild*).

KBUILD_RUSTFLAGS_KERNEL
$(RUSTC) options specific for built-in

$(KBUILD_RUSTFLAGS_KERNEL) contains extra Rust compiler flags used to compile resident kernel code.

KBUILD_RUSTFLAGS_MODULE
Options for $(RUSTC) when building modules

$(KBUILD_RUSTFLAGS_MODULE) is used to add arch-specific options that are used for $(RUSTC).

From commandline RUSTFLAGS_MODULE shall be used (see *Kbuild*).

KBUILD_LDFLAGS_MODULE
Options for $(LD) when linking modules

$(KBUILD_LDFLAGS_MODULE) is used to add arch-specific options used when linking modules. This is often a linker script.

From commandline LDFLAGS_MODULE shall be used (see *Kbuild*).

KBUILD_LDS
The linker script with full path. Assigned by the top-level Makefile.

KBUILD_VMLINUX_OBJS
All object files for vmlinux. They are linked to vmlinux in the same order as listed in KBUILD_VMLINUX_OBJS.

The objects listed in scripts/head-object-list.txt are exceptions; they are placed before the other objects.

KBUILD_VMLINUX_LIBS
All .a `lib` files for vmlinux. KBUILD_VMLINUX_OBJS and KBUILD_VMLINUX_LIBS together specify all the object files used to link vmlinux.

5.7.2 Add prerequisites to archheaders

The archheaders: rule is used to generate header files that may be installed into user space by `make header_install`.

It is run before `make archprepare` when run on the architecture itself.

5.7.3 Add prerequisites to archprepare

The archprepare: rule is used to list prerequisites that need to be built before starting to descend down in the subdirectories.

This is usually used for header files containing assembler constants.

Example:

```
#arch/arm/Makefile
archprepare: maketools
```

In this example, the file target maketools will be processed before descending down in the subdirectories.

See also chapter XXX-TODO that describes how kbuild supports generating offset header files.

5.7.4 List directories to visit when descending

An arch Makefile cooperates with the top Makefile to define variables which specify how to build the vmlinux file. Note that there is no corresponding arch-specific section for modules; the module-building machinery is all architecture-independent.

core-y, libs-y, drivers-y
$(libs-y) lists directories where a lib.a archive can be located.

The rest list directories where a built-in.a object file can be located.

Then the rest follows in this order:

$(core-y), $(libs-y), $(drivers-y)

The top level Makefile defines values for all generic directories, and arch/$(SRCARCH)/Makefile only adds architecture-specific directories.

Example:

```
# arch/sparc/Makefile
core-y                  += arch/sparc/

libs-y                  += arch/sparc/prom/
libs-y                  += arch/sparc/lib/

drivers-$(CONFIG_PM) += arch/sparc/power/
```

5.7.5 Architecture-specific boot images

An arch Makefile specifies goals that take the vmlinux file, compress it, wrap it in bootstrapping code, and copy the resulting files somewhere. This includes various kinds of installation commands. The actual goals are not standardized across architectures.

It is common to locate any additional processing in a boot/ directory below arch/$(SRCARCH)/.

Kbuild does not provide any smart way to support building a target specified in boot/. Therefore arch/$(SRCARCH)/Makefile shall call make manually to build a target in boot/.

The recommended approach is to include shortcuts in arch/$(SRCARCH)/Makefile, and use the full path when calling down into the arch/$(SRCARCH)/boot/Makefile.

Example:

```
#arch/x86/Makefile
boot := arch/x86/boot
bzImage: vmlinux
        $(Q)$(MAKE) $(build)=$(boot) $(boot)/$@
```

`$(Q)$(MAKE) $(build)=<dir>` is the recommended way to invoke make in a subdirectory.

There are no rules for naming architecture-specific targets, but executing `make help` will list all relevant targets. To support this, $(archhelp) must be defined.

Example:

```
#arch/x86/Makefile
define archhelp
  echo  '* bzImage      - Compressed kernel image (arch/x86/boot/bzImage)'
endif
```

When make is executed without arguments, the first goal encountered will be built. In the top level Makefile the first goal present is all:.

An architecture shall always, per default, build a bootable image. In `make help`, the default goal is highlighted with a *.

Add a new prerequisite to all: to select a default goal different from vmlinux.

Example:

```
#arch/x86/Makefile
all: bzImage
```

When `make` is executed without arguments, bzImage will be built.

5.7.6 Commands useful for building a boot image

Kbuild provides a few macros that are useful when building a boot image.

ld
Link target. Often, LDFLAGS_$@ is used to set specific options to ld.

Example:

```
#arch/x86/boot/Makefile
LDFLAGS_bootsect := -Ttext 0x0 -s --oformat binary
LDFLAGS_setup    := -Ttext 0x0 -s --oformat binary -e begtext

targets += setup setup.o bootsect bootsect.o
$(obj)/setup $(obj)/bootsect: %: %.o FORCE
        $(call if_changed,ld)
```

In this example, there are two possible targets, requiring different options to the linker. The linker options are specified using the LDFLAGS_$@ syntax - one for each potential target.

$(targets) are assigned all potential targets, by which kbuild knows the targets and will:

1) check for commandline changes
2) delete target during make clean

The `: %: %.o` part of the prerequisite is a shorthand that frees us from listing the setup.o and bootsect.o files.

Note: It is a common mistake to forget the `targets :=` assignment, resulting in the target file being recompiled for no obvious reason.

objcopy
Copy binary. Uses OBJCOPYFLAGS usually specified in arch/$(SRCARCH)/Makefile.

OBJCOPYFLAGS_$@ may be used to set additional options.

gzip
Compress target. Use maximum compression to compress target.

Example:

```
#arch/x86/boot/compressed/Makefile
$(obj)/vmlinux.bin.gz: $(vmlinux.bin.all-y) FORCE
        $(call if_changed,gzip)
```

dtc
Create flattened device tree blob object suitable for linking into vmlinux. Device tree blobs linked into vmlinux are placed in an init section in the image. Platform code *must* copy the blob to non-init memory prior to calling unflatten_device_tree().

To use this command, simply add `*.dtb` into obj-y or targets, or make some other target depend on `%.dtb`

A central rule exists to create `$(obj)/%.dtb` from `$(src)/%.dts`; architecture Makefiles do no need to explicitly write out that rule.

Example:

```
targets += $(dtb-y)
DTC_FLAGS ?= -p 1024
```

5.7.7 Preprocessing linker scripts

When the vmlinux image is built, the linker script arch/$(SRCARCH)/kernel/vmlinux.lds is used.

The script is a preprocessed variant of the file vmlinux.lds.S located in the same directory.

kbuild knows .lds files and includes a rule `*lds.S -> *lds`.

Example:

```
#arch/x86/kernel/Makefile
extra-y := vmlinux.lds
```

The assignment to extra-y is used to tell kbuild to build the target vmlinux.lds.

The assignment to $(CPPFLAGS_vmlinux.lds) tells kbuild to use the specified options when building the target vmlinux.lds.

When building the `*.lds` target, kbuild uses the variables:

```
KBUILD_CPPFLAGS      : Set in top-level Makefile
cppflags-y           : May be set in the kbuild makefile
CPPFLAGS_$(@F)       : Target-specific flags.
                       Note that the full filename is used in this
                       assignment.
```

The kbuild infrastructure for `*lds` files is used in several architecture-specific files.

5.7.8 Generic header files

The directory include/asm-generic contains the header files that may be shared between individual architectures.

The recommended approach how to use a generic header file is to list the file in the Kbuild file.

See *generic-y* for further info on syntax etc.

5.7.9 Post-link pass

If the file arch/xxx/Makefile.postlink exists, this makefile will be invoked for post-link objects (vmlinux and modules.ko) for architectures to run post-link passes on. Must also handle the clean target.

This pass runs after kallsyms generation. If the architecture needs to modify symbol locations, rather than manipulate the kallsyms, it may be easier to add another postlink target for .tmp_vmlinux? targets to be called from link-vmlinux.sh.

For example, powerpc uses this to check relocation sanity of the linked vmlinux file.

5.8 Kbuild syntax for exported headers

The kernel includes a set of headers that is exported to userspace. Many headers can be exported as-is but other headers require a minimal pre-processing before they are ready for user-space.

The pre-processing does:

- drop kernel-specific annotations
- drop include of compiler.h
- drop all sections that are kernel internal (guarded by `ifdef __KERNEL__`)

All headers under include/uapi/, include/generated/uapi/, arch/<arch>/include/uapi/ and arch/<arch>/include/generated/uapi/ are exported.

A Kbuild file may be defined under arch/<arch>/include/uapi/asm/ and arch/<arch>/include/asm/ to list asm files coming from asm-generic.

See subsequent chapter for the syntax of the Kbuild file.

5.8.1 no-export-headers

no-export-headers is essentially used by include/uapi/linux/Kbuild to avoid exporting specific headers (e.g. kvm.h) on architectures that do not support it. It should be avoided as much as possible.

5.8.2 generic-y

If an architecture uses a verbatim copy of a header from include/asm-generic then this is listed in the file arch/$(SRCARCH)/include/asm/Kbuild like this:

Example:

```
#arch/x86/include/asm/Kbuild
generic-y += termios.h
generic-y += rtc.h
```

During the prepare phase of the build a wrapper include file is generated in the directory:

```
arch/$(SRCARCH)/include/generated/asm
```

When a header is exported where the architecture uses the generic header a similar wrapper is generated as part of the set of exported headers in the directory:

```
usr/include/asm
```

The generated wrapper will in both cases look like the following:

Example: termios.h:

```
#include <asm-generic/termios.h>
```

5.8.3 generated-y

If an architecture generates other header files alongside generic-y wrappers, generated-y specifies them.

This prevents them being treated as stale asm-generic wrappers and removed.

Example:

```
#arch/x86/include/asm/Kbuild
generated-y += syscalls_32.h
```

5.8.4 mandatory-y

mandatory-y is essentially used by include/(uapi/)asm-generic/Kbuild to define the minimum set of ASM headers that all architectures must have.

This works like optional generic-y. If a mandatory header is missing in arch/$(SRCARCH)/include/(uapi/)/asm, Kbuild will automatically generate a wrapper of the asm-generic one.

5.9 Kbuild Variables

The top Makefile exports the following variables:

VERSION, PATCHLEVEL, SUBLEVEL, EXTRAVERSION
: These variables define the current kernel version. A few arch Makefiles actually use these values directly; they should use $(KERNELRELEASE) instead.

  $(VERSION), $(PATCHLEVEL), and $(SUBLEVEL) define the basic three-part version number, such as “2”, “4”, and “0”. These three values are always numeric.

  $(EXTRAVERSION) defines an even tinier sublevel for pre-patches or additional patches. It is usually some non-numeric string such as “-pre4”, and is often blank.

KERNELRELEASE
: $(KERNELRELEASE) is a single string such as “2.4.0-pre4”, suitable for constructing installation directory names or showing in version strings. Some arch Makefiles use it for this purpose.

ARCH
: This variable defines the target architecture, such as “i386”, “arm”, or “sparc”. Some kbuild Makefiles test $(ARCH) to determine which files to compile.

  By default, the top Makefile sets $(ARCH) to be the same as the host system architecture. For a cross build, a user may override the value of $(ARCH) on the command line:

  ```
  make ARCH=m68k ...
  ```

SRCARCH
: This variable specifies the directory in arch/ to build.

  ARCH and SRCARCH may not necessarily match. A couple of arch directories are biarch, that is, a single `arch/*/` directory supports both 32-bit and 64-bit.

For example, you can pass in ARCH=i386, ARCH=x86_64, or ARCH=x86. For all of them, SRCARCH=x86 because arch/x86/ supports both i386 and x86_64.

INSTALL_PATH
This variable defines a place for the arch Makefiles to install the resident kernel image and System.map file. Use this for architecture-specific install targets.

INSTALL_MOD_PATH, MODLIB
$(INSTALL_MOD_PATH) specifies a prefix to $(MODLIB) for module installation. This variable is not defined in the Makefile but may be passed in by the user if desired.

$(MODLIB) specifies the directory for module installation. The top Makefile defines $(MODLIB) to $(INSTALL_MOD_PATH)/lib/modules/$(KERNELRELEASE). The user may override this value on the command line if desired.

INSTALL_MOD_STRIP
If this variable is specified, it will cause modules to be stripped after they are installed. If INSTALL_MOD_STRIP is "1", then the default option --strip-debug will be used. Otherwise, the INSTALL_MOD_STRIP value will be used as the option(s) to the strip command.

5.10 Makefile language

The kernel Makefiles are designed to be run with GNU Make. The Makefiles use only the documented features of GNU Make, but they do use many GNU extensions.

GNU Make supports elementary list-processing functions. The kernel Makefiles use a novel style of list building and manipulation with few `if` statements.

GNU Make has two assignment operators, `:=` and `=`. `:=` performs immediate evaluation of the right-hand side and stores an actual string into the left-hand side. `=` is like a formula definition; it stores the right-hand side in an unevaluated form and then evaluates this form each time the left-hand side is used.

There are some cases where `=` is appropriate. Usually, though, `:=` is the right choice.

5.11 Credits

- Original version made by Michael Elizabeth Chastain, <mailto:mec@shout.net>
- Updates by Kai Germaschewski <kai@tp1.ruhr-uni-bochum.de>
- Updates by Sam Ravnborg <sam@ravnborg.org>
- Language QA by Jan Engelhardt <jengelh@gmx.de>

5.12 TODO

- Describe how kbuild supports shipped files with _shipped.
- Generating offset header files.
- Add more variables to chapters 7 or 9?

CHAPTER

SIX

BUILDING EXTERNAL MODULES

This document describes how to build an out-of-tree kernel module.

6.1 1. Introduction

“kbuild” is the build system used by the Linux kernel. Modules must use kbuild to stay compatible with changes in the build infrastructure and to pick up the right flags to “gcc.” Functionality for building modules both in-tree and out-of-tree is provided. The method for building either is similar, and all modules are initially developed and built out-of-tree.

Covered in this document is information aimed at developers interested in building out-of-tree (or “external”) modules. The author of an external module should supply a makefile that hides most of the complexity, so one only has to type “make” to build the module. This is easily accomplished, and a complete example will be presented in section 3.

6.2 2. How to Build External Modules

To build external modules, you must have a prebuilt kernel available that contains the configuration and header files used in the build. Also, the kernel must have been built with modules enabled. If you are using a distribution kernel, there will be a package for the kernel you are running provided by your distribution.

An alternative is to use the “make” target “modules_prepare.” This will make sure the kernel contains the information required. The target exists solely as a simple way to prepare a kernel source tree for building external modules.

NOTE: “modules_prepare” will not build Module.symvers even if CONFIG_MODVERSIONS is set; therefore, a full kernel build needs to be executed to make module versioning work.

6.3 2.1 Command Syntax

The command to build an external module is:

```
$ make -C <path_to_kernel_src> M=$PWD
```

The kbuild system knows that an external module is being built due to the “M=<dir>” option given in the command.

To build against the running kernel use:

```
$ make -C /lib/modules/`uname -r`/build M=$PWD
```

Then to install the module(s) just built, add the target “modules_install” to the command:

```
$ make -C /lib/modules/`uname -r`/build M=$PWD modules_install
```

6.4 2.2 Options

($KDIR refers to the path of the kernel source directory.)

make -C $KDIR M=$PWD

-C $KDIR
The directory where the kernel source is located. “make” will actually change to the specified directory when executing and will change back when finished.

M=$PWD
Informs kbuild that an external module is being built. The value given to “M” is the absolute path of the directory where the external module (kbuild file) is located.

6.5 2.3 Targets

When building an external module, only a subset of the “make” targets are available.

make -C $KDIR M=$PWD [target]

The default will build the module(s) located in the current directory, so a target does not need to be specified. All output files will also be generated in this directory. No attempts are made to update the kernel source, and it is a precondition that a successful “make” has been executed for the kernel.

modules
The default target for external modules. It has the same functionality as if no target was specified. See description above.

modules_install
Install the external module(s). The default location is /lib/modules/<kernel_release>/extra/, but a prefix may be added with INSTALL_MOD_PATH (discussed in section 5).

clean
Remove all generated files in the module directory only.

help
List the available targets for external modules.

6.6 2.4 Building Separate Files

It is possible to build single files that are part of a module. This works equally well for the kernel, a module, and even for external modules.

Example (The module foo.ko, consist of bar.o and baz.o):

```
make -C $KDIR M=$PWD bar.lst
make -C $KDIR M=$PWD baz.o
make -C $KDIR M=$PWD foo.ko
make -C $KDIR M=$PWD ./
```

6.7 3. Creating a Kbuild File for an External Module

In the last section we saw the command to build a module for the running kernel. The module is not actually built, however, because a build file is required. Contained in this file will be the name of the module(s) being built, along with the list of requisite source files. The file may be as simple as a single line:

```
obj-m := <module_name>.o
```

The kbuild system will build <module_name>.o from <module_name>.c, and, after linking, will result in the kernel module <module_name>.ko. The above line can be put in either a "Kbuild" file or a "Makefile." When the module is built from multiple sources, an additional line is needed listing the files:

```
<module_name>-y := <src1>.o <src2>.o ...
```

NOTE: Further documentation describing the syntax used by kbuild is located in *Linux Kernel Makefiles*.

The examples below demonstrate how to create a build file for the module 8123.ko, which is built from the following files:

```
8123_if.c
8123_if.h
8123_pci.c
8123_bin.o_shipped      <= Binary blob
```

6.7.1 3.1 Shared Makefile

An external module always includes a wrapper makefile that supports building the module using “make” with no arguments. This target is not used by kbuild; it is only for convenience. Additional functionality, such as test targets, can be included but should be filtered out from kbuild due to possible name clashes.

Example 1:

```
--> filename: Makefile
ifneq ($(KERNELRELEASE),)
# kbuild part of makefile
obj-m  := 8123.o
8123-y := 8123_if.o 8123_pci.o 8123_bin.o

else
# normal makefile
KDIR ?= /lib/modules/`uname -r`/build

default:
        $(MAKE) -C $(KDIR) M=$$PWD

# Module specific targets
genbin:
        echo "X" > 8123_bin.o_shipped

endif
```

The check for KERNELRELEASE is used to separate the two parts of the makefile. In the example, kbuild will only see the two assignments, whereas “make” will see everything except these two assignments. This is due to two passes made on the file: the first pass is by the “make” instance run on the command line; the second pass is by the kbuild system, which is initiated by the parameterized “make” in the default target.

6.7.2 3.2 Separate Kbuild File and Makefile

In newer versions of the kernel, kbuild will first look for a file named “Kbuild,” and only if that is not found, will it then look for a makefile. Utilizing a “Kbuild” file allows us to split up the makefile from example 1 into two files:

Example 2:

```
--> filename: Kbuild
obj-m  := 8123.o
8123-y := 8123_if.o 8123_pci.o 8123_bin.o

--> filename: Makefile
KDIR ?= /lib/modules/`uname -r`/build

default:
        $(MAKE) -C $(KDIR) M=$$PWD
```

```
# Module specific targets
genbin:
        echo "X" > 8123_bin.o_shipped
```

The split in example 2 is questionable due to the simplicity of each file; however, some external modules use makefiles consisting of several hundred lines, and here it really pays off to separate the kbuild part from the rest.

The next example shows a backward compatible version.

Example 3:

```
--> filename: Kbuild
obj-m  := 8123.o
8123-y := 8123_if.o 8123_pci.o 8123_bin.o

--> filename: Makefile
ifneq ($(KERNELRELEASE),)
# kbuild part of makefile
include Kbuild

else
# normal makefile
KDIR ?= /lib/modules/`uname -r`/build

default:
        $(MAKE) -C $(KDIR) M=$$PWD

# Module specific targets
genbin:
        echo "X" > 8123_bin.o_shipped

endif
```

Here the “Kbuild” file is included from the makefile. This allows an older version of kbuild, which only knows of makefiles, to be used when the “make” and kbuild parts are split into separate files.

6.7.3 3.3 Binary Blobs

Some external modules need to include an object file as a blob. kbuild has support for this, but requires the blob file to be named <filename>_shipped. When the kbuild rules kick in, a copy of <filename>_shipped is created with _shipped stripped off, giving us <filename>. This shortened filename can be used in the assignment to the module.

Throughout this section, 8123_bin.o_shipped has been used to build the kernel module 8123.ko; it has been included as 8123_bin.o:

```
8123-y := 8123_if.o 8123_pci.o 8123_bin.o
```

Although there is no distinction between the ordinary source files and the binary file, kbuild will pick up different rules when creating the object file for the module.

6.8 3.4 Building Multiple Modules

kbuild supports building multiple modules with a single build file. For example, if you wanted to build two modules, foo.ko and bar.ko, the kbuild lines would be:

```
obj-m := foo.o bar.o
foo-y := <foo_srcs>
bar-y := <bar_srcs>
```

It is that simple!

6.9 4. Include Files

Within the kernel, header files are kept in standard locations according to the following rule:

- If the header file only describes the internal interface of a module, then the file is placed in the same directory as the source files.
- If the header file describes an interface used by other parts of the kernel that are located in different directories, then the file is placed in include/linux/.

  NOTE:
  There are two notable exceptions to this rule: larger subsystems have their own directory under include/, such as include/scsi; and architecture specific headers are located under arch/$(SRCARCH)/include/.

6.9.1 4.1 Kernel Includes

To include a header file located under include/linux/, simply use:

```
#include <linux/module.h>
```

kbuild will add options to “gcc” so the relevant directories are searched.

6.9.2 4.2 Single Subdirectory

External modules tend to place header files in a separate include/ directory where their source is located, although this is not the usual kernel style. To inform kbuild of the directory, use either ccflags-y or CFLAGS_<filename>.o.

Using the example from section 3, if we moved 8123_if.h to a subdirectory named include, the resulting kbuild file would look like:

```
--> filename: Kbuild
obj-m := 8123.o

```

```
ccflags-y := -Iinclude
8123-y := 8123_if.o 8123_pci.o 8123_bin.o
```

Note that in the assignment there is no space between -I and the path. This is a limitation of kbuild: there must be no space present.

6.9.3 4.3 Several Subdirectories

kbuild can handle files that are spread over several directories. Consider the following example:

```
.
|__ src
|   |__ complex_main.c
|   |__ hal
|       |__ hardwareif.c
|       |__ include
|           |__ hardwareif.h
|__ include
|__ complex.h
```

To build the module complex.ko, we then need the following kbuild file:

```
--> filename: Kbuild
obj-m := complex.o
complex-y := src/complex_main.o
complex-y += src/hal/hardwareif.o

ccflags-y := -I$(src)/include
ccflags-y += -I$(src)/src/hal/include
```

As you can see, kbuild knows how to handle object files located in other directories. The trick is to specify the directory relative to the kbuild file's location. That being said, this is NOT recommended practice.

For the header files, kbuild must be explicitly told where to look. When kbuild executes, the current directory is always the root of the kernel tree (the argument to "-C") and therefore an absolute path is needed. $(src) provides the absolute path by pointing to the directory where the currently executing kbuild file is located.

6.10 5. Module Installation

Modules which are included in the kernel are installed in the directory:

/lib/modules/$(KERNELRELEASE)/kernel/

And external modules are installed in:

/lib/modules/$(KERNELRELEASE)/extra/

6.10.1 5.1 INSTALL_MOD_PATH

Above are the default directories but as always some level of customization is possible. A prefix can be added to the installation path using the variable INSTALL_MOD_PATH:

```
$ make INSTALL_MOD_PATH=/frodo modules_install
=> Install dir: /frodo/lib/modules/$(KERNELRELEASE)/kernel/
```

INSTALL_MOD_PATH may be set as an ordinary shell variable or, as shown above, can be specified on the command line when calling "make." This has effect when installing both in-tree and out-of-tree modules.

6.10.2 5.2 INSTALL_MOD_DIR

External modules are by default installed to a directory under /lib/modules/$(KERNELRELEASE)/extra/, but you may wish to locate modules for a specific functionality in a separate directory. For this purpose, use INSTALL_MOD_DIR to specify an alternative name to "extra.":

```
$ make INSTALL_MOD_DIR=gandalf -C $KDIR \
       M=$PWD modules_install
=> Install dir: /lib/modules/$(KERNELRELEASE)/gandalf/
```

6.11 6. Module Versioning

Module versioning is enabled by the CONFIG_MODVERSIONS tag, and is used as a simple ABI consistency check. A CRC value of the full prototype for an exported symbol is created. When a module is loaded/used, the CRC values contained in the kernel are compared with similar values in the module; if they are not equal, the kernel refuses to load the module.

Module.symvers contains a list of all exported symbols from a kernel build.

6.11.1 6.1 Symbols From the Kernel (vmlinux + modules)

During a kernel build, a file named Module.symvers will be generated. Module.symvers contains all exported symbols from the kernel and compiled modules. For each symbol, the corresponding CRC value is also stored.

The syntax of the Module.symvers file is:

```
<CRC>       <Symbol>         <Module>                              <Export␣
↪Type>     <Namespace>

0xe1cc2a05  usb_stor_suspend drivers/usb/storage/usb-storage  EXPORT_
↪SYMBOL_GPL USB_STORAGE
```

The fields are separated by tabs and values may be empty (e.g. if no namespace is defined for an exported symbol).

For a kernel build without CONFIG_MODVERSIONS enabled, the CRC would read 0x00000000.

Module.symvers serves two purposes:

1) It lists all exported symbols from vmlinux and all modules.
2) It lists the CRC if CONFIG_MODVERSIONS is enabled.

6.11.2 6.2 Symbols and External Modules

When building an external module, the build system needs access to the symbols from the kernel to check if all external symbols are defined. This is done in the MODPOST step. modpost obtains the symbols by reading Module.symvers from the kernel source tree. During the MODPOST step, a new Module.symvers file will be written containing all exported symbols from that external module.

6.11.3 6.3 Symbols From Another External Module

Sometimes, an external module uses exported symbols from another external module. Kbuild needs to have full knowledge of all symbols to avoid spitting out warnings about undefined symbols. Two solutions exist for this situation.

NOTE: The method with a top-level kbuild file is recommended but may be impractical in certain situations.

Use a top-level kbuild file
If you have two modules, foo.ko and bar.ko, where foo.ko needs symbols from bar.ko, you can use a common top-level kbuild file so both modules are compiled in the same build. Consider the following directory layout:

```
./foo/ <= contains foo.ko
./bar/ <= contains bar.ko
```

The top-level kbuild file would then look like:

```
#./Kbuild (or ./Makefile):
        obj-m := foo/ bar/
```

And executing:

```
$ make -C $KDIR M=$PWD
```

will then do the expected and compile both modules with full knowledge of symbols from either module.

Use “make” variable KBUILD_EXTRA_SYMBOLS
If it is impractical to add a top-level kbuild file, you can assign a space separated list of files to KBUILD_EXTRA_SYMBOLS in your build file. These files will be loaded by modpost during the initialization of its symbol tables.

6.12 7. Tips & Tricks

6.12.1 7.1 Testing for CONFIG_FOO_BAR

Modules often need to check for certain *CONFIG_* options to decide if a specific feature is included in the module. In kbuild this is done by referencing the *CONFIG_* variable directly:

```
#fs/ext2/Makefile
obj-$(CONFIG_EXT2_FS) += ext2.o

ext2-y := balloc.o bitmap.o dir.o
ext2-$(CONFIG_EXT2_FS_XATTR) += xattr.o
```

External modules have traditionally used “grep” to check for specific *CONFIG_* settings directly in .config. This usage is broken. As introduced before, external modules should use kbuild for building and can therefore use the same methods as in-tree modules when testing for *CONFIG_* definitions.

CHAPTER

SEVEN

EXPORTING KERNEL HEADERS FOR USE BY USERSPACE

The “make headers_install” command exports the kernel’s header files in a form suitable for use by userspace programs.

The linux kernel’s exported header files describe the API for user space programs attempting to use kernel services. These kernel header files are used by the system’s C library (such as glibc or uClibc) to define available system calls, as well as constants and structures to be used with these system calls. The C library’s header files include the kernel header files from the “linux” subdirectory. The system’s libc headers are usually installed at the default location /usr/include and the kernel headers in subdirectories under that (most notably /usr/include/linux and /usr/include/asm).

Kernel headers are backwards compatible, but not forwards compatible. This means that a program built against a C library using older kernel headers should run on a newer kernel (although it may not have access to new features), but a program built against newer kernel headers may not work on an older kernel.

The “make headers_install” command can be run in the top level directory of the kernel source code (or using a standard out-of-tree build). It takes two optional arguments:

```
make headers_install ARCH=i386 INSTALL_HDR_PATH=/usr
```

ARCH indicates which architecture to produce headers for, and defaults to the current architecture. The linux/asm directory of the exported kernel headers is platform-specific, to see a complete list of supported architectures use the command:

```
ls -d include/asm-* | sed 's/.*-//'
```

INSTALL_HDR_PATH indicates where to install the headers. It defaults to “./usr”.

An ‘include’ directory is automatically created inside INSTALL_HDR_PATH and headers are installed in ‘INSTALL_HDR_PATH/include’.

The kernel header export infrastructure is maintained by David Woodhouse <dwmw2@infradead.org>.

CHAPTER

EIGHT

RECURSION ISSUES

8.1 issue #1

```
# Simple Kconfig recursive issue
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
# Test with:
#
# make KBUILD_KCONFIG=Documentation/kbuild/Kconfig.recursion-issue-01␣
 ↪allnoconfig
#
# This Kconfig file has a simple recursive dependency issue. In order to
# understand why this recursive dependency issue occurs lets consider what
# Kconfig needs to address. We iterate over what Kconfig needs to address
# by stepping through the questions it needs to address sequentially.
#
#  * What values are possible for CORE?
#
# CORE_BELL_A_ADVANCED selects CORE, which means that it influences the values
# that are possible for CORE. So for example if CORE_BELL_A_ADVANCED is 'y',
# CORE must be 'y' too.
#
#  * What influences CORE_BELL_A_ADVANCED ?
#
# As the name implies CORE_BELL_A_ADVANCED is an advanced feature of
# CORE_BELL_A so naturally it depends on CORE_BELL_A. So if CORE_BELL_A is 'y'
# we know CORE_BELL_A_ADVANCED can be 'y' too.
#
#   * What influences CORE_BELL_A ?
#
# CORE_BELL_A depends on CORE, so CORE influences CORE_BELL_A.
#
# But that is a problem, because this means that in order to determine
# what values are possible for CORE we ended up needing to address questions
# regarding possible values of CORE itself again. Answering the original
# question of what are the possible values of CORE would make the kconfig
# tools run in a loop. When this happens Kconfig exits and complains about
# the "recursive dependency detected" error.
#
```

```
# Reading the Documentation/kbuild/Kconfig.recursion-issue-01 file it may be
# obvious that an easy to solution to this problem should just be the removal
# of the "select CORE" from CORE_BELL_A_ADVANCED as that is implicit already
# since CORE_BELL_A depends on CORE. Recursive dependency issues are not always
# so trivial to resolve, we provide another example below of practical
# implications of this recursive issue where the solution is perhaps not so
# easy to understand. Note that matching semantics on the dependency on
# CORE also consist of a solution to this recursive problem.

mainmenu "Simple example to demo kconfig recursive dependency issue"

config CORE
        tristate

config CORE_BELL_A
        tristate
        depends on CORE

config CORE_BELL_A_ADVANCED
        tristate
        depends on CORE_BELL_A
        select CORE
```

8.2 issue #2

```
# Cumulative Kconfig recursive issue
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#
# Test with:
#
# make KBUILD_KCONFIG=Documentation/kbuild/Kconfig.recursion-issue-02␣
 ↪allnoconfig
#
# The recursive limitations with Kconfig has some non intuitive implications on
# kconfig semantics which are documented here. One known practical implication
# of the recursive limitation is that drivers cannot negate features from other
# drivers if they share a common core requirement and use disjoint semantics to
# annotate those requirements, ie, some drivers use "depends on" while others
# use "select". For instance it means if a driver A and driver B share the same
# core requirement, and one uses "select" while the other uses "depends on" to
# annotate this, all features that driver A selects cannot now be negated by
# driver B.
#
# A perhaps not so obvious implication of this is that, if semantics on these
# core requirements are not carefully synced, as drivers evolve features
# they select or depend on end up becoming shared requirements which cannot be
# negated by other drivers.
#
```

```
# The example provided in Documentation/kbuild/Kconfig.recursion-issue-02
# describes a simple driver core layout of example features a kernel might
# have. Let's assume we have some CORE functionality, then the kernel has a
# series of bells and whistles it desires to implement, its not so advanced so
# it only supports bells at this time: CORE_BELL_A and CORE_BELL_B. If
# CORE_BELL_A has some advanced feature CORE_BELL_A_ADVANCED which selects
# CORE_BELL_A then CORE_BELL_A ends up becoming a common BELL feature which
# other bells in the system cannot negate. The reason for this issue is
# due to the disjoint use of semantics on expressing each bell's relationship
# with CORE, one uses "depends on" while the other uses "select". Another
# more important reason is that kconfig does not check for dependencies listed
# under 'select' for a symbol, when such symbols are selected kconfig them
# as mandatory required symbols. For more details on the heavy handed nature
# of select refer to Documentation/kbuild/Kconfig.select-break
#
# To fix this the "depends on CORE" must be changed to "select CORE", or the
# "select CORE" must be changed to "depends on CORE".
#
# For an example real world scenario issue refer to the attempt to remove
# "select FW_LOADER" [0], in the end the simple alternative solution to this
# problem consisted on matching semantics with newly introduced features.
#
# [0] https://lore.kernel.org/r/1432241149-8762-1-git-send-email-mcgrof@do-not-
 ↪panic.com

mainmenu "Simple example to demo cumulative kconfig recursive dependency␣
 ↪implication"

config CORE
        tristate

config CORE_BELL_A
        tristate
        depends on CORE

config CORE_BELL_A_ADVANCED
        tristate
        select CORE_BELL_A

config CORE_BELL_B
        tristate
        depends on !CORE_BELL_A
        select CORE
```

CHAPTER

NINE

REPRODUCIBLE BUILDS

It is generally desirable that building the same source code with the same set of tools is reproducible, i.e. the output is always exactly the same. This makes it possible to verify that the build infrastructure for a binary distribution or embedded system has not been subverted. This can also make it easier to verify that a source or tool change does not make any difference to the resulting binaries.

The Reproducible Builds project has more information about this general topic. This document covers the various reasons why building the kernel may be unreproducible, and how to avoid them.

9.1 Timestamps

The kernel embeds timestamps in three places:

- The version string exposed by `uname()` and included in `/proc/version`
- File timestamps in the embedded initramfs
- If enabled via `CONFIG_IKHEADERS`, file timestamps of kernel headers embedded in the kernel or respective module, exposed via `/sys/kernel/kheaders.tar.xz`

By default the timestamp is the current time and in the case of `kheaders` the various files' modification times. This must be overridden using the KBUILD_BUILD_TIMESTAMP variable. If you are building from a git commit, you could use its commit date.

The kernel does *not* use the `__DATE__` and `__TIME__` macros, and enables warnings if they are used. If you incorporate external code that does use these, you must override the timestamp they correspond to by setting the SOURCE_DATE_EPOCH environment variable.

9.2 User, host

The kernel embeds the building user and host names in `/proc/version`. These must be overridden using the KBUILD_BUILD_USER and KBUILD_BUILD_HOST variables. If you are building from a git commit, you could use its committer address.

9.3 Absolute filenames

When the kernel is built out-of-tree, debug information may include absolute filenames for the source files. This must be overridden by including the `-fdebug-prefix-map` option in the KCFLAGS variable.

Depending on the compiler used, the `__FILE__` macro may also expand to an absolute filename in an out-of-tree build. Kbuild automatically uses the `-fmacro-prefix-map` option to prevent this, if it is supported.

The Reproducible Builds web site has more information about these prefix-map options.

9.4 Generated files in source packages

The build processes for some programs under the `tools/` subdirectory do not completely support out-of-tree builds. This may cause a later source package build using e.g. `make rpm-pkg` to include generated files. You should ensure the source tree is pristine by running `make mrproper` or `git clean -d -f -x` before building a source package.

9.5 Module signing

If you enable `CONFIG_MODULE_SIG_ALL`, the default behaviour is to generate a different temporary key for each build, resulting in the modules being unreproducible. However, including a signing key with your source would presumably defeat the purpose of signing modules.

One approach to this is to divide up the build process so that the unreproducible parts can be treated as sources:

1. Generate a persistent signing key. Add the certificate for the key to the kernel source.
2. Set the `CONFIG_SYSTEM_TRUSTED_KEYS` symbol to include the signing key's certificate, set `CONFIG_MODULE_SIG_KEY` to an empty string, and disable `CONFIG_MODULE_SIG_ALL`. Build the kernel and modules.
3. Create detached signatures for the modules, and publish them as sources.
4. Perform a second build that attaches the module signatures. It can either rebuild the modules or use the output of step 2.

9.6 Structure randomisation

If you enable `CONFIG_RANDSTRUCT`, you will need to pre-generate the random seed in `scripts/basic/randstruct.seed` so the same value is used by each build. See `scripts/gen-randstruct-seed.sh` for details.

9.7 Debug info conflicts

This is not a problem of unreproducibility, but of generated files being *too* reproducible.

Once you set all the necessary variables for a reproducible build, a vDSO's debug information may be identical even for different kernel versions. This can result in file conflicts between debug information packages for the different kernel versions.

To avoid this, you can make the vDSO different for different kernel versions by including an arbitrary string of "salt" in it. This is specified by the Kconfig symbol `CONFIG_BUILD_SALT`.

9.8 Git

Uncommitted changes or different commit ids in git can also lead to different compilation results. For example, after executing `git reset HEAD^`, even if the code is the same, the `include/config/kernel.release` generated during compilation will be different, which will eventually lead to binary differences. See `scripts/setlocalversion` for details.

CHAPTER

TEN

GCC PLUGIN INFRASTRUCTURE

10.1 Introduction

GCC plugins are loadable modules that provide extra features to the compiler[1]. They are useful for runtime instrumentation and static analysis. We can analyse, change and add further code during compilation via callbacks[2], GIMPLE[3], IPA[4] and RTL passes[5].

The GCC plugin infrastructure of the kernel supports building out-of-tree modules, cross-compilation and building in a separate directory. Plugin source files have to be compilable by a C++ compiler.

Currently the GCC plugin infrastructure supports only some architectures. Grep “select HAVE_GCC_PLUGINS” to find out which architectures support GCC plugins.

This infrastructure was ported from grsecurity[6] and PaX[7].

--

10.2 Purpose

GCC plugins are designed to provide a place to experiment with potential compiler features that are neither in GCC nor Clang upstream. Once their utility is proven, the goal is to upstream the feature into GCC (and Clang), and then to finally remove them from the kernel once the feature is available in all supported versions of GCC.

Specifically, new plugins should implement only features that have no upstream compiler support (in either GCC or Clang).

When a feature exists in Clang but not GCC, effort should be made to bring the feature to upstream GCC (rather than just as a kernel-specific GCC plugin), so the entire ecosystem can benefit from it.

Similarly, even if a feature provided by a GCC plugin does *not* exist in Clang, but the feature is proven to be useful, effort should be spent to upstream the feature to GCC (and Clang).

[1] https://gcc.gnu.org/onlinedocs/gccint/Plugins.html
[2] https://gcc.gnu.org/onlinedocs/gccint/Plugin-API.html#Plugin-API
[3] https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html
[4] https://gcc.gnu.org/onlinedocs/gccint/IPA.html
[5] https://gcc.gnu.org/onlinedocs/gccint/RTL.html
[6] https://grsecurity.net/
[7] https://pax.grsecurity.net/

After a feature is available in upstream GCC, the plugin will be made unbuildable for the corresponding GCC version (and later). Once all kernel-supported versions of GCC provide the feature, the plugin will be removed from the kernel.

10.3 Files

$(src)/scripts/gcc-plugins

This is the directory of the GCC plugins.

$(src)/scripts/gcc-plugins/gcc-common.h

This is a compatibility header for GCC plugins. It should be always included instead of individual gcc headers.

$(src)/scripts/gcc-plugins/gcc-generate-gimple-pass.h, $(src)/scripts/gcc-plugins/gcc-generate-ipa-pass.h, $(src)/scripts/gcc-plugins/gcc-generate-simple_ipa-pass.h, $(src)/scripts/gcc-plugins/gcc-generate-rtl-pass.h

These headers automatically generate the registration structures for GIMPLE, SIMPLE_IPA, IPA and RTL passes. They should be preferred to creating the structures by hand.

10.4 Usage

You must install the gcc plugin headers for your gcc version, e.g., on Ubuntu for gcc-10:

```
apt-get install gcc-10-plugin-dev
```

Or on Fedora:

```
dnf install gcc-plugin-devel libmpc-devel
```

Or on Fedora when using cross-compilers that include plugins:

```
dnf install libmpc-devel
```

Enable the GCC plugin infrastructure and some plugin(s) you want to use in the kernel config:

```
CONFIG_GCC_PLUGINS=y
CONFIG_GCC_PLUGIN_LATENT_ENTROPY=y
...
```

Run gcc (native or cross-compiler) to ensure plugin headers are detected:

```
gcc -print-file-name=plugin
CROSS_COMPILE=arm-linux-gnu- ${CROSS_COMPILE}gcc -print-file-name=plugin
```

The word “plugin” means they are not detected:

```
plugin
```

A full path means they are detected:

```
/usr/lib/gcc/x86_64-redhat-linux/12/plugin
```

To compile the minimum tool set including the plugin(s):

```
make scripts
```

or just run the kernel make and compile the whole kernel with the cyclomatic complexity GCC plugin.

10.5 4. How to add a new GCC plugin

The GCC plugins are in scripts/gcc-plugins/. You need to put plugin source files right under scripts/gcc-plugins/. Creating subdirectories is not supported. It must be added to scripts/gcc-plugins/Makefile, scripts/Makefile.gcc-plugins and a relevant Kconfig file.

CHAPTER

ELEVEN

BUILDING LINUX WITH CLANG/LLVM

This document covers how to build the Linux kernel with Clang and LLVM utilities.

11.1 About

The Linux kernel has always traditionally been compiled with GNU toolchains such as GCC and binutils. Ongoing work has allowed for Clang and LLVM utilities to be used as viable substitutes. Distributions such as Android, ChromeOS, OpenMandriva, and Chimera Linux use Clang built kernels. Google's and Meta's datacenter fleets also run kernels built with Clang.

LLVM is a collection of toolchain components implemented in terms of C++ objects. Clang is a front-end to LLVM that supports C and the GNU C extensions required by the kernel, and is pronounced "klang," not "see-lang."

11.2 Building with LLVM

Invoke `make` via:

```
make LLVM=1
```

to compile for the host target. For cross compiling:

```
make LLVM=1 ARCH=arm64
```

11.3 The LLVM= argument

LLVM has substitutes for GNU binutils utilities. They can be enabled individually. The full list of supported make variables:

```
make CC=clang LD=ld.lld AR=llvm-ar NM=llvm-nm STRIP=llvm-strip \
  OBJCOPY=llvm-objcopy OBJDUMP=llvm-objdump READELF=llvm-readelf \
  HOSTCC=clang HOSTCXX=clang++ HOSTAR=llvm-ar HOSTLD=ld.lld
```

`LLVM=1` expands to the above.

If your LLVM tools are not available in your PATH, you can supply their location using the LLVM variable with a trailing slash:

```
make LLVM=/path/to/llvm/
```

which will use `/path/to/llvm/clang`, `/path/to/llvm/ld.lld`, etc. The following may also be used:

```
PATH=/path/to/llvm:$PATH make LLVM=1
```

If your LLVM tools have a version suffix and you want to test with that explicit version rather than the unsuffixed executables like `LLVM=1`, you can pass the suffix using the `LLVM` variable:

```
make LLVM=-14
```

which will use `clang-14`, `ld.lld-14`, etc.

To support combinations of out of tree paths with version suffixes, we recommend:

```
PATH=/path/to/llvm/:$PATH make LLVM=-14
```

`LLVM=0` is not the same as omitting `LLVM` altogether, it will behave like `LLVM=1`. If you only wish to use certain LLVM utilities, use their respective make variables.

The same value used for `LLVM=` should be set for each invocation of `make` if configuring and building via distinct commands. `LLVM=` should also be set as an environment variable when running scripts that will eventually run `make`.

11.4 Cross Compiling

A single Clang compiler binary (and corresponding LLVM utilities) will typically contain all supported back ends, which can help simplify cross compiling especially when `LLVM=1` is used. If you use only LLVM tools, `CROSS_COMPILE` or target-triple-prefixes become unnecessary. Example:

```
make LLVM=1 ARCH=arm64
```

As an example of mixing LLVM and GNU utilities, for a target like `ARCH=s390` which does not yet have `ld.lld` or `llvm-objcopy` support, you could invoke `make` via:

```
make LLVM=1 ARCH=s390 LD=s390x-linux-gnu-ld.bfd \
  OBJCOPY=s390x-linux-gnu-objcopy
```

This example will invoke `s390x-linux-gnu-ld.bfd` as the linker and `s390x-linux-gnu-objcopy`, so ensure those are reachable in your `$PATH`.

`CROSS_COMPILE` is not used to prefix the Clang compiler binary (or corresponding LLVM utilities) as is the case for GNU utilities when `LLVM=1` is not set.

11.5 The LLVM_IAS= argument

Clang can assemble assembler code. You can pass `LLVM_IAS=0` to disable this behavior and have Clang invoke the corresponding non-integrated assembler instead. Example:

```
make LLVM=1 LLVM_IAS=0
```

`CROSS_COMPILE` is necessary when cross compiling and `LLVM_IAS=0` is used in order to set `--prefix=` for the compiler to find the corresponding non-integrated assembler (typically, you don't want to use the system assembler when targeting another architecture). Example:

```
make LLVM=1 ARCH=arm LLVM_IAS=0 CROSS_COMPILE=arm-linux-gnueabi-
```

11.6 Ccache

`ccache` can be used with `clang` to improve subsequent builds, (though KBUILD_BUILD_TIMESTAMP should be set to a deterministic value between builds in order to avoid 100% cache misses, see Reproducible_builds for more info):

KBUILD_BUILD_TIMESTAMP='' make LLVM=1 CC="ccache clang"

11.7 Supported Architectures

LLVM does not target all of the architectures that Linux supports and just because a target is supported in LLVM does not mean that the kernel will build or work without any issues. Below is a general summary of architectures that currently work with `CC=clang` or `LLVM=1`. Level of support corresponds to "S" values in the MAINTAINERS files. If an architecture is not present, it either means that LLVM does not target it or there are known issues. Using the latest stable version of LLVM or even the development tree will generally yield the best results. An architecture's `defconfig` is generally expected to work well, certain configurations may have problems that have not been uncovered yet. Bug reports are always welcome at the issue tracker below!

Architecture	Level of support	make command
arm	Supported	`LLVM=1`
arm64	Supported	`LLVM=1`
hexagon	Maintained	`LLVM=1`
loongarch	Maintained	`LLVM=1`
mips	Maintained	`LLVM=1`
powerpc	Maintained	`LLVM=1`
riscv	Supported	`LLVM=1`
s390	Maintained	`CC=clang`
um (User Mode)	Maintained	`LLVM=1`
x86	Supported	`LLVM=1`

11.8 Getting Help

- Website
- Mailing List: <llvm@lists.linux.dev>
- Old Mailing List Archives
- Issue Tracker
- IRC: #clangbuiltlinux on irc.libera.chat
- Telegram: @ClangBuiltLinux
- Wiki
- Beginner Bugs

11.9 Getting LLVM

We provide prebuilt stable versions of LLVM on kernel.org. These have been optimized with profile data for building Linux kernels, which should improve kernel build times relative to other distributions of LLVM.

Below are links that may be useful for building LLVM from source or procuring it through a distribution's package manager.

- https://releases.llvm.org/download.html
- https://github.com/llvm/llvm-project
- https://llvm.org/docs/GettingStarted.html
- https://llvm.org/docs/CMake.html
- https://apt.llvm.org/
- https://www.archlinux.org/packages/extra/x86_64/llvm/
- https://github.com/ClangBuiltLinux/tc-build
- https://github.com/ClangBuiltLinux/linux/wiki/Building-Clang-from-source
- https://android.googlesource.com/platform/prebuilts/clang/host/linux-x86/