Kconfig Language

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.

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.

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.

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 ".

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'.

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.

Kconfig hints

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.

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'.

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_ASM_GOTO
      def_bool $(success,$(srctree)/scripts/gcc-goto.sh $(CC))
```

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).

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.

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
```

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.

Simple Kconfig recursive issue

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

Test with:

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

Cumulative Kconfig recursive issue

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

Test with:

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

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)

commit	fix
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.

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 on 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.

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.

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).

Confirming this could prove useful as Kconfig stands as one of the leading industrial variability modeling languages [1] [2]. 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].

[0] https://www.eng.uwaterloo.ca/~shshe/kconfig_semantics.pdf

[1] (*1*,*2*,*3*) https://gsd.uwaterloo.ca/sites/default/files/vm-2013-berger.pdf

[2] https://gsd.uwaterloo.ca/sites/default/files/ase241-berger_0.pdf

[3] https://gsd.uwaterloo.ca/sites/default/files/icse2011.pdf

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

[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