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".

- 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.
	kconfig should one day warn about such things.

- 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" or "---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.
  "---help---" and "help" do not differ in behaviour, "---help---" is
  used to help visually separate configuration logic from help within
  the file as an aid to developers.

- misc options: "option" <symbol>[=<value>]
  Various less common options can be defined via this option syntax,
  which can modify the behaviour of the menu entry and its config
  symbol. These options are currently possible:

  - "defconfig_list"
    This declares a list of default entries which can be used when
    looking for the default configuration (which is used when the main
    .config doesn't exists yet.)

  - "modules"
    This declares the symbol to be used as the MODULES symbol, which
    enables the third modular state for all config symbols.

  - "env"=<value>
    This imports the environment variable into Kconfig. It behaves like
    a default, except that the value comes from the environment, this
    also means that the behaviour when mixing it with normal defaults is
    undefined at this point. The symbol is currently not exported back
    to the build environment (if this is desired, it can be done via
    another symbol).

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)
           '(' <expr> ')'                       (4)
           '!' <expr>                           (5)
           <expr> '&&' <expr>                   (6)
           <expr> '||' <expr>                   (7)
```

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) Returns the value of the expression. Used to override precedence.
(5) Returns the result of (2-/expr/).
(6) Returns the result of min(/expr/, /expr/).
(7) 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 always
visible when MODULES is visible (the (empty) dependency of MODULES is
also part of the comment dependencies).

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.

choices:

```
	"choice"
	<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, 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.

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.

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.

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'.

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).