【linux-2.6.31】kbuild

Translated By: openspace

Date : 2009-11-25

组织结构

00-INDEX

描述了全文的结构组织

kbuild.txt

描述 kbuild 相关信息，供开发人员参考

kconfig.txt

make *config 使用帮助

kconfig-language.txt

kconfig 文件中使用的配置语言规范

makefiles.txt

描述内核 makefile 文件，供开发人员参考

modules.txt

如何编译并安装模块

kbuild 环境变量

KCPPFLAGS

预处理时可作为附加选项传递。预处理选项用于 kbuild 进行预处理操作的所有场合，包括构建 C 文件和汇编文件

KAFLAGS

汇编器附加选项

KCFLAGS

C 编译器附加选项

KBUILD_VERBOSE

设置 kbuild 输出详细信息。可以通过“V=...”指定相同的值。使用 make help 获取完整列表。设置“V=...”优先级高于 KBUILD_VERBOSE

KBUILD_EXTMOD

设置构建外部的模块时的源代码目录。可以通过多种方式指定目录：

1) 命令行使用“M=...”
2) 环境变量 KBUILD_EXTMOD
3) 环境变量 SUBDIRS

所列出的方式以优先级由高到低的顺序排列。使用“M=...”具有最高优先级

KBUILD_OUTPUT

指定构建内核的输出路径。还可以通过“O=...”来指定。“O=...”的优先级要高于 KBUILD_OUTPUT

ARCH

设置 ARCH 为要构建的目标体系结构。多数情况下目录体系结构的名字与目录 arch/ 下的名字相同。但是一些体系结构例如 x86 和 sparc 使用别名：

x86: i386 for 32 bit, x86_64 for 64 bit
sparc: sparc for 32 bit, sparc64 for 64 bit

CROSS_COMPILE

指定 binutils 文件名的任意固定部分；CROSS_COMPILE 可以是文件名的一部分或者完整的路径。在一些操作中 CROSS_COMPILE 也用于 ccache

CF

很少使用的附加选项。GF 常以如下方式用在命令行中：

make CF=-Wbitwise C=2

INSTALL_PATH

INSTALL_PATH 指定了放置更新的内核以及 system map 映像的位置。缺省为/boot；可以把它设置为其他值

MODLIB

指定安装模块的位置。缺省值为：

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

可以忽略缺省值而指定新的值

INSTALL_MOD_PATH

指定构建时模块路径 MODLIB 的前缀。makefile 文件中没有定义该值，可以通过传递给 make 的参数指定

INSTALL_MOD_STRIP

如果定义该变量，模块在安装之后会进行 strip 操作。如果 INSTALL_MOD_STRIP 为 1，那么缺省的选项—strip-debug 被激活；否则 INSTALL_MOD_STRIP 用作 strip 命令的选项

INSTALL_FW_PATH

指定 firmware blobs 的安装路径。缺省值为：

$(INSTALL_MOD_PATH)/lib/firmware

可以设置新的值

INSTALL_HDR_PATH

指定执行“make headers_*”安装的用户空间头文件的存放路径。缺省值为：

$(objtree)/usr

$(objtree)是存放输出文件的目录。可以在命令行通过“O=...”指定输出目录。可以设置新的值

KBUILD_MODPOST_WARN

设置该变量可以避免在最后的模块链接阶段出现未定义符号的错误，会将错误转换为警告

KBUILD_MODPOST_NOFINAL

设置该变量可以跳过最后的模块链接。用于加快测试编译

KBUILD_EXTRA_SYMBOLS

用于使用其他模块的符号的模块。查看 modules.txt 获取更多信息

ALLSOURCE_ARCHS

对于 tags/TAGS/cscope，可以指定在数据库中保护多个体系结构的信息，这些体系结构之间用空格分隔。例如：

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

kconfig

本篇所包含的信息有助于使用“make *config”。

“使用 make help”可以得到所有配置方式的列表。xconfig('qconf')和 menuconfig ('mconf')的说明也包含在帮助文本中。在浏览、搜索或者使用其它文档时务必参考本文档。

==========

概述

新的内核发行版通常会引入新的配置符号。更重要的，新的内核版本可能会改变配置符号的名字。这种情况发生时，使用先前生成的.config 文件并使用“make oldconfig”不足以构建新的工作内核，这就需要查看引入了哪些新的内核符号。

使用“make oldconfig”时要查看新配置符号的列表，使用

```
cp user/some/old.config .config
yes "" | make oldconfig > conf.new
```

config 程序会给出新符号列表，新符号的值还不明确。当然，.config 文件也更新为包含了新的（缺省）值，所以可以使用：

```
grep "(NEW)" conf.new
```

来查看新的配置符号，或者使用 diff 来查看新旧.config 文件的不同：

```
diff .config.old .config | less
```

（显然，我们需要更方便的工具。）

*config 相关环境变量

KCONFIG_CONFIG

该环境变量用于指定缺省的内核配置文件名，来取代缺省的“.config”

KCONFIG_OVERWRITECONFIG

如果设置 KCONFIG_OVERWRITECONFIG，.config 为一个符号链接时 Kconfig 不会打破该符号链接

KCONFIG_NOTIMESTAMP

如果该变量不为空，生成的.config 文件中的时间戳信息被忽略

{allyes/allmod/allno/rand}config 相关环境变量

KCONFIG_ALLCONFIG

（部分描述基于 lkml 中 Rob Landley 的邮件 re: miniconfig）

---

allyesconfig/allmodconfig/allnoconfig/randconfig 可以将环境变量 KCONFIG_

ALLCONFIG 用作包含用户要求设置为特定值的配置符号的一个标志或一个文件名。如果使用 KCONFIG_ALLCONFIG 时没有用文件名，“make *config”会查看名为“all{yes/mod/no/random}.config”（对应于使用的*config 命令）的文件，以获取强制设置的符号值。如果没有找到该文件，会查看名为“all.config”的文件以获取强制设置的符号值。

这样你可以创建“miniature”配置（miniconfig），或者定制包含感兴趣的配置符号的配置文件。然后内核配置系统生产完整的.config 文件，其中包括你的 miniconfig 文件中的符号。

KCONFIG_ALLCONFIG 文件包含（通常为子集）预设的配置符号。这些变量的设置还是要进行依赖性检测。

例如：

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

或者

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

或者

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

这些例子会禁止多数选项（allnonconfig），并根据指定的 mini-config 文件的显示指定禁止或者激活某些选项

silentoldconfig 相关环境变量

KCONFIG_NOSILENTUPDATE

如果该变量值不为空，会阻止静默更新内核配置（需要显示更新）

KCONFIG_AUTOCONFIG

设置该变量可以指定“auto.conf”文件的路径和名字。缺省值为“include/config/auto.conf”

KCONFIG_AUTOHEADER

设置该变量可以指定“autoconf.h”（头）文件。缺省值为“include/linux/autoconf.h”

==========

menuconfig

搜索 CONFIG 符号

在 menuconfig 中搜索：

函数 Search 搜索内核配置符号名，所以需要知道与搜索目标相近的信息。

例如：

```
/hotplug
```

会列出所有包含“hotplug”的配置符号，例如 HOTPLUG、HOTPLUG_CPU、MEMORY_HOTPLUG。

使用搜索帮助，输入/加 TAB-TAB-TAB（会高亮<Help>）并回车。这时会告诉你可以在搜索字符串中使用正则表达式 regexes），所以如果对 MEMORY_HOTPLUG 不感兴趣，可以试试

```
/^hotplug
```

menuconfig 的用户界面选项

MENUCONFIG_COLOR

使用该选项可以选择不同的颜色主题。要选择一个主题：

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

可选主题有：

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)

MENUCONFIG_MODE

该模式将所有的子菜单显示在一个大的树形结构下。例如：

```
make MENUCONFIG_MODE=single_menu menuconfig
```

==========

xconfig

在 xconfig 中搜索：

函数 Search 搜索内核配置符号名，所以需要知道与搜索目标相近的信息。例如：

```
Ctrl-F hotplug
```

或者

```
Menu: File, Search, hotplug
```

会列出所有包含“hotplug”的配置符号表项。在搜索对话框中，可以改变任何非灰色表项的配置设置。而且不需要返回主菜单就可以输入不同的搜索字符串。

==========

gconfig

在 gconfig 中搜索：

None（gconfig 并不像 xconfig 或者 menuconfig 那样被维护着）；但是 gconfig 相比 xconfig 有更多的界面选择。

kconfig-language

简介

配置数据库是以树形结构组织的一组配置选项：

```
+- 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
+- ...
```

每个表项都有自己的依赖关系。这些依赖关系用于决定某个表项是否可见。只有父表项可见，才能看到子表项。

菜单表项

大多数表项都定义了一个配置选项；其他表项构成其基础。单个配置选项的定义类似如下方式：

```
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.  ...
```

以关键字开始的行后面可以跟多个参数。“config”标志一个配置选项的开始。接下来几行定义了该配置选项的属性。属性可以为配置选项的类型、输入提示、依赖关系、帮助文本和缺省值。使用同一个名字可以对一个配置属性定义多次，但是每个定义只能有一个输入提示，而且类型不能冲突。

菜单属性

一个菜单表项可以有多个属性。不是所有的属性都适用于任何场合（参考语法）。

- 类型定义：**"bool"/"tristate"/"string"/"hex"/"int"**

每个配置选项必须有一个类型。只有两种基本类型：tristate 和 string；其他类型以该两种类型为基础。类型定义可以接受一个输入提示，所以下面两个例子是等

价的：

```
bool "Networking support"
```

和

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

- 输入提示：**"prompt" <prompt> ["if" <expr>]**

每个菜单表项最多有一个提示，用于显示给用户。用于该提示的依赖关系可以通过"if"添加

- 缺省值：**"default" <expr> ["if" <expr>]**

一个配置选项可以有多个缺省值。如果有多个可见的缺省值，则只有第一个是活跃的。缺省值不限于定义菜单项的地方；这意味着可以在其他地方定义缺省值，或者事先定义后覆盖

缺省值只在用户没有设置（通过上面的输入提示）的时候赋予配置符号。如果输入提示可见，那么缺省值会显示给用户，用户可以设置新的值

缺省值的依赖关系可以通过"if"添加

- 类型定义 + 缺省值：**"def_bool"/"def_tristate" <expr> ["if" <expr>]**

这是类型定义和值的简洁表示。缺省值的依赖关系可以通过"if"添加

- 依赖关系：**"depends on" <expr>**

这定义了该表项的一个依赖关系。如果定义多个依赖关系，要通过'&&'来组织。依赖关系适用于该表项的所有其他属性（通过"if"表达式添加），所以下面两个例子是等价的：

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

和

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

- 反向依赖：**"select" <symbol> ["if" <expr>]**

普通的依赖关系限制了一个符号的上限（看下面），反向依赖则限制另一个符号的下限。当前菜单符号的值可以看做<symbol>的最小值。如果多次选择<symbol>，以最大值为下限

反向依赖只能用于 boolean 或者 tristate 类型的符号

注意：谨慎使用 select。select 会不查看依赖关系就为符号设置值。乱用 select 可以在没有设置 FOO 依赖的 BAR 时选择符号 FOO。通常 select 只用于不可见的符号（没有提示）和没有依赖关系的符号。这会限制使用，但是会避免非法配置。总有一天 kconfig 会对这些设置发出警告

- 数值范围：**"range" <symbol> <symbol> ["if" <expr>]**

可以限制 int 和 hex 符号的输入值的范围。用户输入的值只能大于等于第一个符号，并且小于等于第二个符号

- 帮助文本：**"help"或者"---help---"**

定义帮助文本。帮助文本的结束有缩进程度来决定，这意味着在遇到比帮助文本的第一行有缩进的行时终止

"---help---"和"help"没有什么不同，"---help---"用于视觉上帮助开发人员区分配置逻辑和帮助信息

- 杂乱选项：**"option" <symbol>[=<value>]**

可以通过该语法定义不太常用的选项，这些选项会影响菜单项的行为和配置符号。当前这类选项包括：

- **"defconfig_list"**

声明缺省表项列表，在查找缺省配置（此时还没有.config 文件）时会用到这些表项

- **"modules"**

声明 MODULES 使用的符号，使用该选项可以激活配置符号的第三种模块状态

- **"env"=<value>**

将环境变量导入 Kconfig。除非使用了环境中的值，否则符号的值是缺省的；这意味着将普通缺省值混在一起使用，结果是未定义的

当前不能向构建环境导出符号（需要的话可以通过另一个符号来做到这一点）

菜单的依赖关系

依赖关系定义了一个菜单项的可视性，并限制了 tristate 符号的输入范围。表达式中使用的 tristate 逻辑使用了多个状态来描述模块状态，而不是普通的布尔逻辑。依赖表达式的语法如下：

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

表达式按照优先级由高到低的顺序排列：

(1) 将符号转换为表达式。boolean 和 tristate 符号转换为对应的表达式值，其它符号转

换为'n'

(2) 如果两个符号值相等，则返回'y'，否则返回'n'

(3) 如果两个符号的值相等，则返回'n'，否则返回'y'

(4) 返回表达式的值，用于覆盖优先级

(5) 返回(2-/expr/)的结果

(6) 返回 min(/expr/, /expr/)的结果

(7) 返回 max(/expr/, /expr/)的结果

一个表达式的值可能是'n'、'm'或'y'（或者 0、1、2）。当表达式计算为'm'或者'y'时菜单项可见。

有两种符号：常量和非常量符号。多数符号为非常量符号，通过'config'语句定义；非常量符号由字母、数字或者下划线构成。常量符号是表达式的一部分。常量符号由单引号或者双引号括起来；在引号内，可以放置任何字符，而引号本身要用'\'转义。

菜单的结构

可以通过两种方式指定菜单项在树结构中的位置。第一种为显示指定：

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

config NETDEVICES
        ...

endmenu
```

所有在"menu" ... "endmenu"块中的表项成为"Network device support"的子表项。所有的子表项继承从菜单项得到的依赖关系，例如，这表示依赖"NET"会添加到配置选项 NETDEVICES 的依赖关系列表中。

另一种方式是通过分析依赖关系生成菜单结构 。如果某个菜单项通过某种方式依赖于前一个表项，那么该菜单项就成为它的一个子表项。首先，前一个符号必须在依赖关系列表中，然后要满足下面两个条件中的一个：

- 如果 parent 设为'n'，子表项必须变为不可见
- 如果 parent 可见，子表项只能是可见的

```
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 直接依赖于 MODULES，所以只有在 MODULES 不为'n'的时候 MODVERSIONS 才可见；反过来说，如果 MODULES 可见则 MODVERSIONS 总是可见的（（空）依赖 MODULES 是 comment 的依赖关系的一部分）。

Kconfig 语法

配置文件描述了一系列菜单表项，每行以关键字开始（除去帮助文本）。下面的关键字表示一个菜单项的结束：

- config
- menuconfig
- choice/endchoice
- comment
- menu/endmenu
- if/endif
- source

前 5 个也是一个菜单项定义的开始。

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

定义一个配置符号<symbol>，可以以上面任意属性为选项。

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

类似于上面的 config 表项，但是提供了前端提示，即所有的子选项应该作为单独的选项列表显示。

```
choices:
        "choice"
        <choice options>
        <choice block>
        "endchoice"
```

定义选择组，可以以上面任意属性为选项。选择项的类型可以为 bool 或者 tristate，boolean 选择项只允许选择一个配置表项，tristate 选择项允许将任意数量的配置项设置为'm'。可用于对应单个硬件存在多个驱动程序而只能将一个驱动程序编译/加载进内核的情况，但是所有的驱动可以编译成模块的形式。

一个选择项还可以有"optional"这个选项，这样可将选择项设置为'n'，也就不用进行任何选择了。

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

定义一个注释，在配置过程中会显示给用户，并会输出到输出文件。只能以依赖关系作为选项。

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

定义一个菜单块，查看“菜单结构”获取更多信息。只能以依赖关系作为选项。

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

定义一个 if 块。依赖表达式<expr>附加到所有的子菜单项中。

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

读取指定的配置文件；总是会对文件进行解析。

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

如果配置程序选中该选项，则设置为配置程序的标题栏。

Kconfig 技巧

这里描述一组 Kconfig 技巧，乍一看并不明显，而实际上多数已经成为 Kconfig 文件中的常见用法。

添加更多特征并使可配置

~~~~~~~~~~~~~~~~~~~~

这是实现与一些而不是全部体系结构相关的特征/功能的常见用法。推荐在通用 Kconfig 文件中使用名为 HAVE_*的配置变量，并选择相关的体系结构。

以通用的 IOMAP 功能为例。

在 lib/Kconfig 中可以看到：

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

config GENERIC_IOMAP
        depends on HAVE_GENERIC_IOMAP && FOO
```

在 lib/Makefile 中可以看到：

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

对于任何使用通用的 IOMAP 功能的体系结构可以看到：

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

注意：使用现有的配置选项来避免创建一个新的配置变量以选择 HAVE_GENERIC_IOMAP

注意：引入内部的配置变量 HAVE_GENERIC_IOMAP，会突破 select 的限制，而 select 会强制将配置选项设置为'y'而不管有什么样的依赖关系。依赖关系转移到符号 GENERIC_IOMAP，这样避免 select 强制将符号设置为'y'的情况

只编译为模块

~~~~~~~~~~~~

限制一个组件只编译为模块，可以将其配置符号设置为"depends on m"。例如：

```
config FOO
        depends on BAR && m

limits FOO to module (=m) or disabled (=n)
```
~~~~~~~~~~~~

makefiles

Linux Kernel Makefiles　　本文描述了 Linux 内核的 Makefile 文件。

=== 目录

=== 1 概述

Makefile 文件包含 5 部分：

Makefile	顶层的 Makefile
.config	内核配置文件
arch/$(ARCH)/Makefile	体系结构的 Makefile
scripts/Makefile.*	适用于所有 kbuild Makefile 的通用规则等
kbuild Makefiles	大约有 500 个这样的文件

顶层 Makefile 读取内核配置操作产生的.config 文件。

顶层 Makefile 构建两个主要的目标：vmlinux（内核映像）和 modules（所有模块文件）。它通过递归访问内核源码树下的子目录来构建这些目标。访问哪些子目录取决于内核配置。顶层 Makefile 包含一个体系结构的 Makefile，用 arch/$(ARCH)/Makefile 指定。体系结构 Makefile 文件为顶层 Makefile 提供了特定体系结构的信息。

每个子目录各有一个 kbuild Makefile 文件来执行从上层传递下来的命令。kbuil Makefile 文件利用.config 文件中的信息来构造由 kbuild 构建内建或者模块对象使用的各种文件列表。

scripts/Makefile.*包含所有的定义/规则，等等。这些信息用于使用 kbuild Makefile 文件来构建内核。

=== 2 谁使用 kbuild Makefile

人们和内核 Makefile 文件之间有 4 种不同的关系。

*Users*负责构建内核。这些人敲入命令，例如"make menuconfig"或者"make"。他们通常不会阅读或者编辑内核 Makefile 文件（或者是其他源文件）。

*Normal developers*研究某一部分，例如设备驱动程序、文件系统和网络协议。他们需要维护所负责的子系统的 kbuild Makefile 文件。为有效地做到这一点，他们需要全面了解内核的 Makefile 文件，以及用于 kbuild 的公共接口信息。

*Arch developers*研究某个整体架构，例如 sparc 或者 ia64。体系结构开发人员需要了解体系结构的 Makefile 以及 kbuild Makefile 文件。

*Kbuild developers*研究对象为内核构建系统本身。他们需要了解内核 Makefile 文件的方方面面。

本文档的读者为 normal developers 和 arch developers。

=== 3 kbuild 文件

内核中多数 Makefile 是基于 kbuild 基础架构的 kbuild Makefile 文件。本章节将描述 kbuild Makefile 中的语法知识。

kbuild 文件最好起名为'Makefile'，但是如果'Makefile'和'Kbuild'文件同时存在的话会使用'Kbuild'文件。

3.1 节“定义目标”是一个简介，后面的章节会提供更加详细的信息和实际的例子。

--- 3.1 定义目标

目标定义是 kbuild Makefile 中的主要部分（核心）。这些行定义了要构建的文件、特殊编译选项以及要递归访问的子目录。

多数简单的 kbuild Makefile 文件包含一行。示例：

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

这告诉 kbuild 目录中有一个名为 foo.o 的对象，该对象根据 foo.c 或者 foo.S 构建。

如果要将 foo.o 构建成模块，要使用 obj-m。因此常使用下面的模式。示例：

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

$(CONFIG_FOO)计算结果可能为 y（内建）或者 m（模块）。如果 CONFIG_FOO 既不是 y 也不是 m，那么不会编译或者链接该文件。

--- 3.2 内建对象 - obj-y

kbuild Makefile 在$(obj-y)列表中指定了用于构建 vmlinux 的对象文件。这些列表依赖于内核配置。

kbuild 编译所有的$(obj-y)文件；然后调用"$(LD) -r"将这些文件合并进一个 built-in.o 文件。稍后父 Makefile 会将 built-in.o 链接进 vmlinux。

$(obj-y)的顺序很重要。允许重复的情况：第一个会链接进 built-in.o，后面的会忽略掉。

链接顺序很重要，因为某些函数（module_init() / __initcall）在启动时会按照顺序来调用。因此时刻记着改变链接顺序，例如可能改变检测 SCSI 控制器的顺序，这样磁盘会重新编号。

示例：

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

--- 3.3 可加载模块 - obj-m

$(obj-m)指定了要构建为可加载的内核模块对象文件。

一个模块可能依赖于一个或多个源文件构建而成。使用一个源文件时，kbuild Makefle 可以简单地将文件添加到$(obj-m)。

示例：

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

注意：在这个例子中$(CONFIG_ISDN_PPP_BSDCOMP)值为'm'

如果一个内核模块依赖多个源文件，要使用类似上面的方式来指定要编译一个模块。

Kbuild 需要知道模块的依赖文件，这需要通过设置变量$(<module_name>-objs)来说明。

示例：

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

在这个例子中，模块名为 isdn.o。Kbuild 会编译$(isdn-objs)中列出的对象，然后对这些文件执行"$(LD) -r"以生成 isdn.o。

Kbuild 通过后缀-objs 和-y 可以识别出构建复合对象所需要的对象。这样，Makefile 可以利用 CONFIG_符号的值来确定是否一个对象要用于构建一个复合对象。

示例：

```
#fs/ext2/Makefile
obj-$(CONFIG_EXT2_FS)        += ext2.o
ext2-y    := balloc.o bitmap.o
ext2-$(CONFIG_EXT2_FS_XATTR) += xattr.o
```

在这个例子中，如果$(CONFIG_EXT2_FS_XATTR)值为'y'，则 xattr.o 只用于构建 ext2.o。

注意：当然，如果将对象编译进内核，上面的语法也行得通。所以，如果 CONFIG_EXT2_FS=y，那么 Kbuild 会按你所期望的那样，生成 ext2.o 文件，然后将其链接到 built-in.o 中。

--- 3.4 导出符号的对象

对于导出符号的模块在 Makefile 中不需要特殊的标识。

--- 3.5 库文件 - lib-y

obj-*列出的目标用于构建模块，或者用于连接到对应特定目录的 built-in.o 文件中；还可以列出会包含在 lib.a 库中的一些对象。Lib-y 列出的所有对象会连接成对应目录的单个库。通过 obj-y 和 lib-y 同时列出的对象不会包含进库中，因为它们可总是可以访问的。同样的情况，lib-m 中的对象要包含在 lib.a 库中。

注意，一个 kbuild makefile 可以同时列出要编译进内核的文件与要编译成库的文件。所以，在一个目录里可以同时存在 built-in.o 与 lib.a 两个文件。

示例：

```
#arch/i386/lib/Makefile
lib-y    := checksum.o delay.o
```

该 makefile 将基于 checksum.o 和 delay.o 来创建一个库文件 lib.a。要让 kbuild 真正意识这里需要构建一个库文件 lib.a，需要将对应目录加到 libs-y 列表中。可以参考“6.3 访问子目录时列出要访问的目录”。

对 lib-y 的使用限制于目录 lib/和 arch/*/lib 之内。

--- 3.6 递归向下访问子目录

一个 Makefile 负责编译所在目录的对象。在子目录中的文件的编译要由子目录自己的 makefile 来管理。只要你让 kbuild 知道它应该递归操作，那么该系统就会在子目录中自动的调用 make 进行递归操作。

要做到这一点，需要使用 obj-y 和 obj-m。ext2 放在一个单独的目录中，fs/中的 Makefile 通过下面的赋值操作告诉 kbuild 需要递归访问子目录。

示例：

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

如果 CONFIG_EXT2_FS 设置为'y' (built-in)或'm' (modular)，对应的变量 obj-也会定下来，并且 kbuild 会访问 ext2 子目录。这些信息只是告诉 kbuild 它需要访问的目录，而子目录中的 Makefile 负责指定要编译的模块以及内建的对象。

将 CONFIG_变量设置成目录名是一个好的习惯。这样对于对应的 CONFIG_值不是'y'和'm'的目录 kbuild 可以直接跳过。

--- 3.7 编译标志

ccflags-y, asflags-y and ldflags-y

这三个变量只在定义的 kbuild Makefile 中有意义。它们用于递归构建中的普通的 cc、as 和 ld 的调用中。注意：先前具有同样的作用的标志名字是：**EXTRA_CFLAGS、EXTRA_AFLAGS 和 EXTRA_LDFLAGS**；现在还支持这些变量，但是不推荐再使用。

ccflags-y 指定使用$(CC)编译 C 文件时的选项。示例：

```
# drivers/sound/emu10k1/Makefile
ccflags-y += -I$(obj)
ccflags-$(DEBUG) += -DEMU10K1_DEBUG
```

该变量是必须的，因为顶层 Makefile 定义了变量$(KBUILD_CFLAGS)并将其用作整个源码树的编译标志。

asflags-y 的作用类似，用于编译汇编文件的目录中使用的选项。示例：

```
#arch/x86_64/kernel/Makefile
asflags-y := -traditional
```

ldflags-y 用于每个目录中调用$(LD)的选项。示例：

```
#arch/m68k/fpsp040/Makefile
ldflags-y := -x
```

subdir-ccflags-y, subdir-asflags-y

这两个标志类似于 ccflags-y 何 as-falgs-y。不同之处在于 subdir-变量会影响它们出现的 kbuild 文件和所有子目录。通过 subdir-*指定的选项会加到命令行上，放在 non-subdir 变量指定的选项之前。

示例：

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

CFLAGS_$@, AFLAGS_$@

CFLAGS_$@和 AFLAGS_$@只用于当前 kbuild Makefile 中的命令。

$(CFLAGS_$@)指定了$(CC)处理每个文件时使用的选项。$@部分对应的值指定了适用的文件。

示例：

```
# drivers/scsi/Makefile
CFLAGS_aha152x.o =   -DAHA152X_STAT -DAUTOCONF
CFLAGS_gdth.o  = # -DDEBUG_GDTH=2 -D__SERIAL__ -D__COM2__ \
                   -DGDTH_STATISTICS
CFLAGS_seagate.o =   -DARBITRATE -DPARITY
                   -DSEAGATE_USE_ASM
```

这三行指定了对应 aha152x.o、gdth.o 和 seagate.o 的编译标志。

$(AFLAGS_$@) 的作用类似，使用与汇编语言源文件。

示例：

```
# arch/arm/kernel/Makefile
AFLAGS_head-armv.o := -DTEXTADDR=$(TEXTADDR) -traditional
AFLAGS_head-armo.o := -DTEXTADDR=$(TEXTADDR) -traditional
```

--- 3.9 跟踪依赖性

kbuild 跟踪下列依赖性：

1) 所有需要的文件(*.c 和*.h)
2) 所有事先需要的文件中使用的 CONFIG_选项
3) 编译目标的命令行

因此，如果改变$(CC)的一个选项，受影响的文件会进行重新编译。

--- 3.10 特殊规则

特殊规则用于 Kbuild 基础架构不能提供所要求的支持的场合。一个典型的例子就是在构建过程中生成的头文件。另一个例子就是特定体系结构的 Makefile 需要采用特殊规则来准备启动镜像，等等。

特殊规则的写法与普通 Make 规则一样。Kbuild 并不在 Makefile 所在的目录中执行，所以所有的特殊规则都要提供参与编译的文件和目标文件的相对路径。

在定义特殊规则时要使用以下两个变量：

$(src)

$(src)是一个指定 Makefile 所在目录的相对路径。总是使用$(src)定位源码树中的文件。

$(obj)

$(obj)是一个指定存放目标的目录的相对路径。总是使用$(obj)定位生成的

文件。

示例：

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

这是一个特殊规则，使用 make 所要求的普通语法。目标文件依赖于两个文件。用$(obj)来定位目标文件，用$(src)来定位源文件（因为它们不是我们生成的文件）。

$(kecho)

向用户显示规则信息是一个好习惯，但是当执行"make -s"时只能看到警告/错误信息。定义$(kecho)会将其后的文本输出到标准输出流，除非使用了"make -s"。

示例：

```
#arch/blackfin/boot/Makefile
$(obj)/vmImage: $(obj)/vmlinux.gz
        $(call if_changed,uimage)
        @$(kecho) 'Kernel: $@ is ready'
```

--- 3.11 $(CC)支持的功能

内核可能由多个不同版本的$(CC)编译，而每个版本都支持一组不同的功能和选项。kbuild 提供了检查$(CC)可用选项的基本功能。通常$(CC)是 gcc 编译器，但也可以使用其它编译器来代替 gcc。

as-option

当编译汇编文件(*.S)时，as-option 用来检查$(CC)是否支持给定选项。如果第一个选项不支持的话，可以用第二个选项。

示例：

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

在上面的例子里，如果$(CC)支持，cflags-y 的值会是选项-Wa$(comma)-isa=$(isa-y)。第二个参数是可选的，当第一个参数不支持时，就会使用该值。

ld-option

当链接目标文件时，用 ld-option 来检查$(CC)是否支持给定选项。如果第一个选项不支持的话，可以用可选的第二个选项来指定。

示例：

```
#arch/i386/kernel/Makefile
vsyscall-flags += $(call ld-option, -Wl$(comma)--hash-style=sysv)
```

在上面的例子中，如果$(CC)支持，ld-option 的值会是选项-Wl$(comma)--hash-

style=sysv，第二个参数是可选的，当第一个参数不支持时，就会使用该值。

as-instr

as-instr 检测是否汇编器会报告特定的指令并输出 option1 或者 option2，测试指令支持 C 转义符。注意：as-instr-option 使用$(AS)选项 KBUILD_CFLAGS。

cc-option

cc-option 用来检查$(CC)是否支持给定选项，并且不支持可选的第二个选项。

示例：

```
#arch/i386/Makefile
cflags-y += $(call cc-option,-march=pentium-mmx,-march=i586)
```

在上面的例子中，如果$(CC)支持，cc-option 的值会是选项-march=pentium-mmx，否则为-march-i586。cc-option 的第二个参数是可选的；如果忽略的话，当第一个选项不被支持时，cflags-y 不会被赋值。注意：cc-option 使用$(CC)选项 KBUILD_CFLAGS。

cc-option-yn

cc-option-yn 用来检查 gcc 是否支持给定选项，支持则返回'y'，否则为'n'。

示例：

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

在上面的例子里，如果$(CC)支持，则$(biarch)设置为选项-m32。当$(biarch)为 y 时，扩展的变量$(aflags-y)和$(cflags-y)就会被分别赋值为-a32 和-m32。注意：cc-option-yn 使用$(CC)选项 KBUILD_CFLAGS。

cc-option-align

gcc 大于 3.0 的版本改变了指定函数、循环等对齐的选项的类型。当使用$(cc-option-align)作为对齐选项的前缀时，会选择正确的前缀：

```
gcc < 3.00
        cc-option-align = -malign
gcc >= 3.00
        cc-option-align = -falign
```

示例：

```
KBUILD_CFLAGS += $(cc-option-align)-functions=4
```

在上面的例子中，选项-falign-funcions=4 被用在 gcc >= 3.00 的情况；对于 gcc < 3.00 的情况， 使用-malign-funcions=4。注意：cc-option-align 使用$(CC)选项 KBUILD_CFLAGS。

cc-version

cc-version 返回$(CC)编译器版本号数字表示。其格式是<major><minor>，二者都是数字。比如，gcc 3.41 会返回 0341。当特定版本的$(CC)在某方面有缺陷时，cc-version 就会很有用；比如，选项-mregparm=3 虽然会被 gcc 接受，但对于某些版本其实现是有问题的。

示例：

```
#arch/i386/Makefile
cflags-y += $(shell \
if [ $(call cc-version) -ge 0300 ] ; then \
        echo "-mregparm=3"; fi ;)
```

在上面的例子中，-mregparm=3 只会在 gcc 的版本号大于等于 3.0 的情况下使用。

cc-ifversion

cc-ifversion 测试$(CC)的版本号，如果版本表达式为真，就赋值为最后的参数。

示例：

```
#fs/reiserfs/Makefile
ccflags-y := $(call cc-ifversion, -lt, 0402, -O1)
```

在这个例子中，如果$(CC)的版本小于 4.2，EXTRA_CFLAGS 就被赋值为-O1。cc-ifversion 可使用所有的 shell 操作符：-eq、-ne、-lt、-le、-gt 和-ge。第三个参数可以像上面例子一样是个文本，但也可以是个扩展的变量或宏。

cc-fullversion

cc-fullversion 用于需要需要 gcc 的精确版本的情况。一个典型的应用是当 GCC 版本被打乱的情况。cc-fullversion 所给出的版本信息比 cc-version 更详细。

示例：

```
#arch/powerpc/Makefile
$(Q)if test "$(call cc-fullversion)" = "040200" ; then \
        echo -n '*** GCC-4.2.0 cannot compile the 64-bit powerpc ' ; \
        false ; \
fi
```

在上面的例子中，对于一个特定的 GCC 版本，编译出错时会向用户输出解释为什么终止的信息。

cc-cross-prefix

cc-cross-prefix 用于检测是否存在一个使用指定前缀的$(CC)。返回 PATH 中第一个匹配的 prefix$(CC)——如果没有匹配则什么也不返回。多个前缀通过 cc-cross-prefix 调用时使用单个空格来分隔。该功能使用于体系结构 Makefile 文件，这种情况下需要将 CROSS_COMPILE 设置为已知的值，而可能需要从多个值中作出选择。推荐仅在交叉编译（主机体系结构不同于目标机体系结构）

时设置 CROSS_COMPILE。如果已经设置了 CROSS_COMPILE，则不要改变它的值。

示例：

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

=== 4 本机程序的支持

kbuild 支持在编译阶段在本机上构建使用的可执行文件。为了使用一个可执行文件，要将编译分成二个阶段。

第一阶段是告诉 kbuild 存在哪些可执行文件。这是通过变量 hostprogs-y 来完成的。

第二阶段是添加一个对可执行文件的显示依赖。有两种方法：在规则中添加依赖，或是利用变量 $(always)。下面会对两种情况进行了详细描述。

--- 4.1 简单本机程序

有时候需要在编译内核时编译并运行一个程序。下面这行就告诉了 kbuild 程序 bin2hex 应该在本机上编译。

示例：

```
hostprogs-y := bin2hex
```

在上面的例子中，kbuild 假设 bin2hex 是由一个与 Makefile 在同一目录下的名为 bin2hex.c 的 C 语言源文件编译而成的。

--- 4.2 复合本机程序

本机程序可以由多个对象文件编译而成。定义本机程序复合对象所使用的语法与内核的相应语法很相似。$(<executeable>-objs)列出了链接成最终可执行文件所需的所有目标文件。

示例：

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

扩展名为.o 的文件是从相应的.c 文件编译而来的。在上面的例子中，checklist.c 编译为 checklist.o，lxdialog.c 编译为 lxdialog.o。最后，两个.o 文件链接成可执行文件 lxdialog。注意：语法<executable>-y 不能用来生成本机程序。

--- 4.3 定义共享库

扩展名为.so 的文件称为共享库，被编译成位置无关对象。kbuild 也支持共享库，但共享库的使用受到限制。在下面的例子中，libconfig.so 共享库用来链接可执行文件

conf 中。

示例：

```
#scripts/kconfig/Makefile
hostprogs-y     := conf
conf-objs       := conf.o libkconfig.so
libkconfig-objs := expr.o type.o
```

共享库文件通常需要一个相应的-objs，在上面的例子中，共享库 libkconfig 是由 expr.o 和 type.o 两个文件组成的。expr.o 和 type.o 将被编译成位置无关代码，然后链接成共享库文件 libkconfig.so。不支持 C++编写的共享库。

--- 4.4 使用 C++编写本机程序

kbuild 也支持用 C++编写本机程序。在此专门介绍是为了支持 kconfig，并且在一般情况下不推荐使用。

示例：

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

在上面的例子中，可执行文件是由 C++文件 qconf.cc 编译而成的，由$(qconf-cxxobjs)来标识。

如果 qconf 是由.c 和.cc 一起编译而成的，那么就需要专门来标识这些文件了。

示例：

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

--- 4.5 控制本机程序的编译器选项

当编译本机程序时，有可能使用到特殊选项。程序总是利用$(HOSTCC)编译，其选项通过$(HOSTCFLAGS)变量指定。可通过使用变量 HOST_EXTRACFLAGS 设置影响所有在 Makefile 文件中要创建的本机程序。

示例：

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

为一个文件设置特定选项，可采用下列形式：

示例：

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

也可以给链接器指定额外选项。

示例：

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

当链接 qconf 时，将会向链接器传递附加选项 "-L$(QTDIR)/lib"。

--- 4.6 什么时候真正构建本机程序

Kbuild 只在被依赖时才编译本机程序。有两种方法来指定：

(1) 在一条规则中显示列出所需要的文件

示例：

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

目标$(obj)/devlist.h 是不会在$(obj)/gen-devlist 更新之前编译的。注意在该规则中所有有关本机程序的引用必须以$(obj)开头。

(2) 使用$(always)

当 Makefile 要编译主机程序，但没有适合的规则时，使用$(always)。

示例：

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

这会告诉 kbuild，即使没有在规则中引用也要编译 lxdialog。

--- 4.7 使用 hostprogs-$(CONFIG_FOO)

kbuild 文件中的一个典型的模式如下：

示例：

```
#scripts/Makefile
hostprogs-$(CONFIG_KALLSYMS) += kallsyms
```

kbuild 知道'y'是编译进内核，而'm'是编译成模块。所以，如果配置符号的值是'm'，kbuild 仍然会编译它。换句话说，kbuild 处理 hostprogs-m 的方式与处理 hostprogs-y 的方式是完全一致的。只是，如果不涉及 CONFIG，最好用 hostprogs-y。

=== 5 kbuild 清理系统的基础结构

"make clean"删除在编译内核时生成的绝大多数目标文件，包括生成的其它文件，例如本机程序。kbuild 通过列表$(hostprogs-y)、$(hostprogs-m)、$(always)、$(extra-y)和$(targets)获知所要编译的目标。这些目标文件都会被"make clean"删除。"make clean"还会删除匹配"*.[oas]"、"*.ko"的文件，以及由 kbuild 额外生成的文件。

通过 kbuild Makefile 中的$(clean-files)可以指定额外的文件。

示例：

```
#drivers/pci/Makefile
clean-files := devlist.h classlist.h
```

当执行"make clean"时，"devlist.h classlist.h"这两个文件将被删除。如果不使用绝对路径（路径以'/'开头）的话，kbuild 假设所要删除的文件与 Makefile 在同一个相对路径上。

要删除一个目录，使用如下操作：

示例：

```
#scripts/package/Makefile
clean-dirs := $(objtree)/debian/
```

这会删除目录 debian，包括其所有的子目录。如果不使用绝对路径（路径以'/'开头）的话，kbuild 假设所要删除的目录与 Makefile 在同一个相对路径上。

一般情况下，kbuild 会递归访问"obj-* := dir/"下的子目录，但有的时候，在体系机构 Makefile 中，kbuild 架构还不足以描述所有的情况，这要显式的指明所要访问的子目录。

示例：

```
#arch/i386/boot/Makefile
subdir- := compressed/
```

上面的赋值指令告诉 kbuild，当执行"make clean"时，要递归访问目录 compressed/。

为了支持在最终编译完成启动镜像后的架构清理工作，还有一个可选的目标 archclean：

示例：

```
#arch/i386/Makefile
archclean:
        $(Q)$(MAKE) $(clean)=arch/i386/boot
```

当"make clean"执行时，make 会递归访问并清理 arch/i386/boot。在 arch/i386/boot 中的 Makefile 使用 subdir-技巧提示 kbuild 进行递归操作。

注解 1：arch/$(ARCH)/Makefile 不能使用"subdir-"，因为该 Makefile 被包含在顶层的 Makefile 中，kbuild 不能在此处进行操作。

注解 2："make clean"会访问 core-y、libs-y、drivers-y 和 net-y 中列出的所有目录。

=== 6 Makefile 基础架构

在递归访问目录之前，顶层 Makefile 要完成环境设置以及递归访问的准备工作。顶层 Makefile 包含公共部分，而 arch/$(ARCH)/Makefile 包含着针对特定体系架构的配置信息。因此，kbuild 要处理特定体系结构，需要在 arch/$(ARCH)/Makefile 中设置一组变量并定义一些目标。

kbuild 执行的几个步骤（大致）如下：

1) 配置内核配置，生成文件.config

2) 将内核的版本号存储在 include/linux/version.h

3) 生成指向 include/asm-$(ARCH)的符号链接 include/asm

4) 更新编译目标所需的文件：

——附加的依赖文件由 arch/$(ARCH)/Makefile 指定

5) 递归向下访问所有在变量 init-* core* drivers-* net-* libs-*中列出的目录，并编译生成目标文件

——这些变量的值可以在 arch/$(ARCH)/Makefile 中扩充

6) 链接所有的对象文件，在对象文件树顶层目录中生成 vmlinux。最先链接的是在 head-y 中列出的文件，该变量由 arch/$(ARCH)/Makefile 设定

7) 最后，特定体系架构进行必须的后续处理并生成最终的启动镜像

——包含生成引导记录

——准备 initrd 镜像或类似文件

--- 6.1 设置变量以调整针对体系架构的构建过程

LDFLAGS 通用的$(LD)选项

该选项在每次调用链接器时都会用到；通常只用来指明模型。

示例：

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

注意：ldflags-y 可用来进一步定制选项。参考章节 3.7。

LDFLAGS_MODULE 链接模块时的$(LD)选项

LDFLAGS_MODULE 所设置的选项将在链接器在链接模块文件.ko 时使用。默认值为"-r"，指定输出文件是可重定位的。

LDFLAGS_vmlinux 链接 vmlinux 时的$(LD)选项

LDFLAGS_vmlinux 用来指定连接最终的 vmlinux 映像时连接器使用的额外选项。LDFLAGS_vmlinux 需要 LDFLAGS_$@的支持。

示例：

```
#arch/i386/Makefile
LDFLAGS_vmlinux := -e stext
```

OBJCOPYFLAGS objcopy 标志

当用$(call if_changed,objcopy)来转换一个.o 文件时，该选项就会被使用。$(call if_changed,objcopy)经常被用来为 vmlinux 生成原始的二进制代码。

示例：

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

#arch/s390/boot/Makefile
$(obj)/image: vmlinux FORCE
```

```
$(call if_changed,objcopy)
```

在这个例子中，二进制文件$(obj)/image 是 vmlinux 的一个二进制版本。$(call if_chagned,xxx)的用法稍后描述。

KBUILD_AFLAGS $(AS)汇编器选择

默认值在顶层 Makefile 中，可以针对具体的体系架构扩充或修改。

示例：

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

KBUILD_CFLAGS $(CC)编译器选项

默认值在顶层 Makefile 中，可以针对具体的体系架构扩充或修改。

通常 KBUILD_CFLAGS 变量的设置依赖与内核的配置。

示例：

```
#arch/i386/Makefile
cflags-$(CONFIG_M386) += -march=i386
KBUILD_CFLAGS += $(cflags-y)
```

许多体系架构的 Makefile 都通过动态的运行目标 C 编译器来检测其所支持的选项：

```
#arch/i386/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)
...
```

第一个例子使用了一个技巧，即当被选择时配置选项扩展为'y'。

CFLAGS_KERNEL 用于内建对象的$(CC)选项

$(CFLAGS_KERNEL)包含了用于编译常驻内核代码的附加编译器选项。

CFLAGS_MODULE 用于模块的$(CC)选项

$(CFLAGS_MODULE)包含了用于编译可装载模块的附加编译器选项。

--- 6.2 将所需文件添加到 archprepare:

规则 archprepare:用于在递归访问子目录之前列出编译目标文件所需文件。通常用于包含汇编常量的头文件。

示例：

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

在这个例子中，目标文件 maketools 将在递归访问子目录之前编译。在 TODO 一章可以看到，kbuild 是如何支持生成 offset 头文件的。

--- 6.3 递归下降访问时列出目录

由体系架构 makefile 和顶层 Makefile 一起来定义变量以指定如何生成 vmlinux 文件。注意，体系架构 Makefile 是不会定义与模块相关的内容的，所有构建模块的定义是与架构无关的。

head-y, init-y, core-y, libs-y, drivers-y, net-y

$(head-y)列出了最先被链接进 vmlinux 的目标文件。$(libs-y)列出生成 lib.a 所在的目录。其余所列的目录是 built-in.o 所在的目录。

$(init-y)放置在$(head-y)之后。余下的顺序如下：$(core-y)、$(libs-y)、$(drivers-y)和$(net-y)。

顶层 makefile 定义了通用的部分，arch/$(ARCH)/Makefile 添加了特定架构的信息。

示例：

```
#arch/sparc64/Makefile
core-y += arch/sparc64/kernel/
libs-y += arch/sparc64/prom/ arch/sparc64/lib/
drivers-$(CONFIG_OPROFILE)  += arch/sparc64/oprofile/
```

--- 6.4 特定体系结构的引导映像

特定体系架构 Makefile 的目的就是生成并压缩 vmlinux 文件，将其放入启动代码，并将其拷贝到适当的位置。这就包含了多种不同的安装命令。多个平台上的实际目标并没有标准化。

通常将附加的处理命令放在 arch/$(ARCH)/下的 boot 目录。

kbuild 并没有为构造 boot/下的目标任何更好的方法。所以 arch/$(ARCH)/Makefile 需要手工调用 make 以构造 boot/下的目标文件。

推荐做法是在 arch/$(ARCH)/Makefile 中包含快捷方式，并在 arch/$(ARCH)/boot/Makefile 中使用完整路径。

示例：

```
#arch/i386/Makefile
boot := arch/i386/boot
bzImage: vmlinux
        $(Q)$(MAKE) $(build)=$(boot) $(boot)/$@
```

"(Q)(MAKE) $(build)=<dir>"是在子目录中调用 make 的推荐做法。

并没有对特定架构目标的命名规则，但执行"make help"可以列出所有的相关目标。

要支持"make help"，必须定义$(archhelp)。

示例：

```
#arch/i386/Makefile
define archhelp
   echo  '* bzImage      - Image (arch/$(ARCH)/boot/bzImage)'
endif
```

当不带参数执行 make 时，会构建第一个目标。在顶层 Makefile 中，第一个目标就是 all。体系架构的 Makefile 默认构造一个可引导的镜像文件。在"make help"中，默认目标就是被加亮的'*'。添加新的依赖文件到 all 就可以选择不同于 vmlinux 的默认目标。

示例：

```
#arch/i386/Makefile
all: bzImage
```

不带参数执行 make 时，bzImage 将被构造出来。

--- 6.5 构建 non-kbuild 目标

extra-y

extra-y 指定了在当前目录下除去通过 obj-*之外所要创建的附加文件。

用 extra-y 指定目标主要是两个目的：

1) 让 kbuild 检查命令行的变化
   - 当调用$(call if_changed,xxx)的时候
2) 让 kbuild 知道哪些文件要在"make clean"时删除

示例：

```
#arch/i386/kernel/Makefile
extra-y := head.o init_task.o
```

在这个例子中，extra-y 用来列出所有只编译但不链接到 built-in.o 的目标文件。

--- 6.6 用于构建引导映像的命令

kbuild 提供了几个用于构建引导镜像的宏。

if_changed

if_changed 后面命令的基础。

用法：

```
target: source(s) FORCE
        $(call if_changed,ld/objcopy/gzip)
```

当执行该规则时，检查是否有文件需要更新，或者在上次调用以后命令行是否发生了改变。如果针对可执行程序的选项发生了变化，后者会进行重新构造。只有在$(targets)列出的的目标文件才能使用 if_changed，否则命令行的检查会失败，并且目标总会被重建。$(targets)的赋值没有前缀$(obj)/。if_changed 可

用来连接自定义的 kbuild 命令，关于 kbuild 自定义命令请看 6.7 节"自定义 kbuild 命令"。

注意：忘记 FORCE 依赖是一种典型的错误。还有一种常见错误是，空格有的时候是有意义的；比如下面的命令就会出错（注意在逗号后面的那个多余的空格）：

```
        target: source(s) FORCE
#WRONG!#   $(call if_changed, ld/objcopy/gzip)
```

ld

链接目标；通常使用 LDFLAGS_$@来设置 ld 的特殊选项。

objcopy

拷贝二进制代码。使用通常在 arch/$(ARCH)/Makefile 中指定的 OBJCOPYFLAGS。OBJCOPYFLAGS_$@可以用来设置附加选项。

gzip

压缩目标文件。尽可能的压缩目标文件。

示例：

```
#arch/i386/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)
```

在这个例子中，有两个可能的目标文件，分别要求不同的链接选项。使用 LDFLAGS_$@语法指定链接器的选项——每个目标一个。$(targets)被赋给所有的目标，kbuild 会识别出哪些是目标，并且还会：

1) 检查命令行是否改变
2) 在"make clean"时，删除目标文件

依赖中的": %: %.o"部分使我们不必列出文件 setup.o 和 bootsect.o。

注意：一个常见错误是忘记给"target :="赋值，导致在目标文件总是无缘无故的被重新编译。

--- 6.7 自定义 kbuild 命令

当 kbuild 的变量 KBUILD_VERBOSE 为 0 时，只会显示命令的简写。如果要为自定义命令使用这一功能，需要设置 2 个变量：

```
quiet_cmd_<command>   - 要显示的命令
cmd_<command>         - 要执行的命令
```

示例：

```
#
quiet_cmd_image = BUILD   $@
     cmd_image = $(obj)/tools/build $(BUILDFLAGS) \
                          $(obj)/vmlinux.bin > $@

targets += bzImage
$(obj)/bzImage: $(obj)/vmlinux.bin $(obj)/tools/build FORCE
          $(call if_changed,image)
          @echo 'Kernel: $@ is ready'
```

当用"make KBUILD_VERBOSE=0"更新$(obj)/bzImage 目标时会显示下面一行：

```
BUILD    arch/i386/boot/bzImage
```

--- 6.8 预处理链接器脚本

构建 vmlinux 映像的时候，会用到链接器脚本 arch/$(ARCH)/kernel/vmlinux.lds。该脚本是当前目录下 vmlinux.lds.S 文件的一个经过预处理的变种。kbuild 识别.lds 文件并加入规则*lds.S -> *lds。

示例：

```
#arch/i386/kernel/Makefile
always := vmlinux.lds

#Makefile
export CPPFLAGS_vmlinux.lds += -P -C -U$(ARCH)
```

$(always)的值是用来让 kbuild 构造目标 vmlinux.lds。$(CPPFLAGS_vmlinux.lds)会让 build 在构造目标 vmlinux.lds 时使用指定选项。

当构造*.lds 目标时，kbuild 要用到下列变量：

KBUILD_CPPFLAGS : 在顶层 Makefile 中设置

cppflags-y : 可能在 kbuild Makefile 中设置

CPPFLAGS_$(@F) : 特定目标选项。注意赋值中要用完整的文件名

针对*.lds 文件的 kbuild 基础架构还被用在许多特定体系结构的文件中。

=== 7 Kbuild 的导出头文件语法

内核包含一组要导出到用户空间的头文件。多数头文件可以直接导出，而有些头文件需要稍做预处理后才能导出。

预处理完成下列工作：

- 去掉内核用注释
- 去掉对 compiler.h 的 include
- 去掉所有内核内部使用的部分（通过 ifdef __KERNEL__指定）

每个相关目录都有一个名为"Kbuild"的文件，该文件指定了要导出的头文件。下面描述了 Kbuild 文件的语法。

--- 7.1 header-y

header-y 指定要导出的头文件

示例：

```
#include/linux/Kbuild
header-y += usb/
header-y += aio_abi.h
```

一个好的习惯是每个文件占用一行，最好以字母顺序排列

header-y 还指定了要查看的子目录。一个子目录通过结尾的'/'标识，上面的例子中为 usb 子目录

在访问父目录之前会先访问子目录

--- 7.2 objhdr-y

objhdr-y 指定要导出的生成文件。生成文件比较特殊，在使用'make O=...'进行构建时需要从另一个目录中查找

示例：

```
#include/linux/Kbuild
objhdr-y += version.h
```

--- 7.3 destination-y

某个体系结构中有一些头文件需要导出到不同的目录，这时需要使用 destination-y。 destination-y 为其所在文件那个目录下的所有要导出的头文件指定了目标目录

示例：

```
#arch/xtensa/platforms/s6105/include/platform/Kbuild
destination-y := include/linux
```

上面的例子中，Kbuild 文件中所有要导出的头文件在导出时会放到目录 "include/linux"下

--- 7.4 unifdef-y (deprecated)

不推荐使用 unifdef-y；用 header-y 替换

=== 8 Kbuild 变量

顶层的 Makefile 导出下列变量：

VERSION, PATCHLEVEL, SUBLEVEL, EXTRAVERSION

这些变量定义了当前内核版本。实际上，一些体系结构内部的 Makefile 直接使用了这些变量；它们应该使用$(KERNELRELEASE)

$(VERSION)、$(PATCHLEVEL)和$(SUBLEVEL)定义了基本的 3 部分版本号，例如"2"、"4"和"0"。这三个变量的值总是数字类型

$(EXTRAVERSION)定义了 pre-patches 或者附加补丁用的更细微的表示。通常为一些非数字的字符串，例如"-pre4"；多数情况下为空

KERNELRELEASE

$(KERNELRELEASE)是类似于"2.4.0-pre4"的单个字符串，用于构建安装目录名或者显示版本。一些体系结构的 Makefile 文件使用该变量

ARCH

该变量定义了目标体系结构，例如"i386"、"arm"或者"sparc"。一些 kbuild Makefile 文件通过检测$(ARCH)来确定要编译的文件

缺省，顶层 Makefile 将$(ARCH)设置为主机体系结构。交叉编译时，用户可以通过命令行为$(ARCH)指定新的值：

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

INSTALL_PATH

该变量定义了体系结构的 Makefile 文件使用的用于安装内核映像和 System.map 文件的位置。用于不同体系结构指定安装目录

INSTALL_MOD_PATH, MODLIB

$(INSTALL_MOD_PATH)指定了用于模块安装的$(MODLIB)的前缀。该变量没有在 Makefile 中定义，但是可以在需要的时候由用户指定

$(MODLIB)指定了模块安装路径。顶层 Makefile 将$(MODLIB)定义为 $(INSTALL_MOD_PATH)/lib/modules/$(KERNELRELEASE)。需要的话用户可以通过命令行参数指定新值

INSTALL_MOD_STRIP

如果指定了该变量，则模块在安装后进行 strip 操作。如果 INSTALL_MOD_ STRIP 为'1'，那么会使用缺省选项—strip-debug；否则 INSTALL_MOD_STRIP 会作为 strip 命令使用的选项

=== 9 Makefile 语言

内核 Makefile 文件是用于 GNU Make 的。这些 Makefile 文件中仅仅用到 GNU Make 文档里描述的功能，但是使用了许多 GNU 扩展。

GNU Make 支持基本的列表处理功能。内核 Makefile 文件使用了一种新式风格列表，列表通过一些"if"语句来建立和操作。

GNU Make 有两种赋值操作符":="和"="。":="会立即执行右边的计算，并将一个实际的字符串赋给左边。"="就像一个公式的定义；它将未计算的式子赋给左边，然后每次使用时计算这个式子。

一些情况下适合使用"="；但是多数情况下":="是最合适的选择。

=== 10 致谢

最初版本由 Michael Elizabeth Chastain <mailto:mec@shout.net>构建；Kai Germaschewski <kai@tp1.ruhr-uni-bochum.de>和 Sam Ravnborg <sam@ravnborg.org>对其进行了更新；Jan Engelhardt <jengelh@gmx.de>负责语言上的错误。

=== 11 TODO

- 描述 kbuild 如何支持使用_shipped 来打包文件
- 生成 offset 头文件
- 在第 7 节添加更多的变量描述？

modules

在这个文档里你可以找到以下信息：

——如何编译外部模块
——如何利用 kbuild 基础结构来编译你的模块
——kbuild 如何安装模块
——如何将模块安装到非标准目录

=== 目录

=== 1. 简介

kbuild 的功能包括编译内核源码树内部和外部的模块。后者通常被称为外部模块或者

"out-of-tree"模块，并且既用在开发过程中，也用于表示没有打算把它放到内核树中的模块。

该文件提供了模块开发人员需要的主要信息。外部模块的作者必须提供一个 Makefile 文件来隐藏复杂的细节，这样只要输入"make"就可以编译这个模块。第 4 章"为外部模块创建 kbuild 文件"提供了一个完整的例子。

=== 2. 如何编译外部模块

kbuild 提供了编译外部模块的功能，但是需要有一个预先编译好的内核。构建内核时可以目标的一个子集可用于外部模块的编译。

--- 2.1 编译外部模块

使用下面的命令编译外部模块：

```
make -C <path-to-kernel> M=`pwd`
```

对于工作内核使用下列方式：

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

上面的命令要操作成功，内核构建时必须设定支持模块。

安装刚构建好的模块：

```
make -C <path-to-kernel> M=`pwd` modules_install
```

稍后会提供更加复杂的例子，上面的可以作为一个入口。

--- 2.2 可用的 target

$KDIR 表示内核源码树的顶层目录对应的路径

make -C $KDIR M=`pwd`

用于构建当前目录下的模块。所有的输出文件将与模块源文件放在同一个目录下。不会修改内核源码，而且要求事先编译内核已经成功

make -C $KDIR M=`pwd` modules

隐含模块对象，看起来好像没有指定目标。参考上面的描述

make -C $KDIR M=`pwd` modules_install

安装外部模块。缺省的安装目录为/lib/modules/<kernel-version>/extra，但是可以通过前缀 INSTALL_MOD_PATH 指定不同的目录——参考其他章节

make -C $KDIR M=`pwd` clean

删除模块中所有生成的文件——不会影响内核源码树

make -C $KDIR M=`pwd` help

列出构建外部模块可用的目标

--- 2.3 可用选项

$KDIR 表示内核源码树的顶层目录对应的路径

make -C $KDIR

用于指定内核源码树的位置。'$KDIR'表示内核源码目录。执行 make 时会先切换到指定目录，完成后再切换回来

make -C $KDIR M=`pwd`

M=用于告诉 kbuild 正在构建一个外部模块。选项 M=指定了外部模块（kbuild 文件）的位置。当编译外部模块时，只能使用常用选项的一个子集

make -C $KDIR SUBDIRS=`pwd`

类似 M=；语法 SUBDIRS=用于向后兼容

--- 2.4 准备编译模块用的内核源码树

要确保内核包含编译外部模块所需的信息，必须使用目标'modules_prepare'。'modules_prepare'仅用作准备内核源码树以支持编译内核模块的简单方式使用。注意：即便设置了 CONFIG_MODVERSIONS，modules_prepare 也不会编译 Module.symvers。因此要支持内核模块的版本化需要进行一个完整的内核构建过程

--- 2.5 从多个文件构建模块

可以分别构建组成模块的单个文件。这种方式适用于内核、模块或者外部模块。

示例(module foo.ko, consist of bar.o, baz.o)：

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

=== 3. 命令使用示例

这个例子展示了在当前内核中构建外部模块时如何使用命令。在下面的例子中，假定发行版支持将内核编译的输出文件放到不同与内核源码树的目录下——如果源码树和输出文件放于同一个目录下例子也可以执行成功。

```
# Kernel source
/lib/modules/<kernel-version>/source -> /usr/src/linux-<version>

# Output from kernel compile
/lib/modules/<kernel-version>/build -> /usr/src/linux-<version>-up
```

切换到存放 kbuild 文件的目录并执行下面的命令来构建模块：

```
cd /home/user/src/module
make -C /usr/src/`uname -r`/source          \
     O=/lib/modules/`uname-r`/build       \
     M=`pwd`
```

然后，使用下面的命令安装模块：

```
make -C /usr/src/`uname -r`/source          \
        O=/lib/modules/`uname-r`/build      \
        M=`pwd`                             \
        modules_install
```

如果认真看一下，你会发现这与上面的命令相同——列出详细目录。

上面的命令很长，下一章会描述一些技巧，可以使过程变得容易一些。

=== 4. 为外部模块创建 kbuild 文件

kbuild 是内核的构建系统，外部模块必须使用 kbuild，这样才能兼容构建系统中的变化，并能使用 gcc 时采用正确的选项，等等。

输入的 kbuild 文件采用 Documentation/kbuild/makefiles.txt 中描述的语法。本章会描述一些技巧，以便处理外部模块。

下面会为使用下列文件的模块创建一个 Makefile：

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

--- 4.1 模块和内核共享的 Makefile

外部模块总是包含一个 Makefile 以支持不带参数的'make'。使用的 Makefile 很可能会包含额外的功能，例如测试目标等等，而且这部分会从 kbuild 中过滤掉，因为如果有命名冲突会影响到 kbuild。

示例 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

KERNELDIR := /lib/modules/`uname -r`/build
all::
        $(MAKE) -C $(KERNELDIR) M=`pwd` $@

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

endif
```

示例 1 中，通过检查 KERNELRELEASE 可以将 Makefile 分成两部分。Kbuild 只会查看两个赋值，而 make 会查看除去两个赋值外的其他指令。

较新的内核中，kbuild 会查看名为 Kbuild 的文件，然后才会查看 Makefile 文件。利用 Kbuild 文件可以将示例 1 中的 Makefile 分成示例 2 所示的两个文件：

示例 2：

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

--> filename: Makefile
KERNELDIR := /lib/modules/`uname -r`/build
all::
	$(MAKE) -C $(KERNELDIR) M=`pwd` $@

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

示例 2 中，我们使用了两个简单的文件，而且简单文件的划分存在一些问题。但是一些外部模块使用数百行的 Makefile，这里值得将 kbuild 部分与其他部分分开。示例 3 给出了一个向后兼容的版本。

示例 3：

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

--> filename: Makefile
ifneq ($(KERNELRELEASE),)
include Kbuild
else
# Normal Makefile

KERNELDIR := /lib/modules/`uname -r`/build
all::
	$(MAKE) -C $(KERNELDIR) M=`pwd` $@

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

endif
```

这里的技巧是从 Makefile 中包含 Kbuild 文件，所以如果一个老版本的 kbuild 使用了 Makefile，Kbuild 文件就会包含进来。

--- 4.2 包含在模块中的二进制 blob

有的外部模块会包含一个.o 作为 blob。kbuild 支持该功能，但是要求 blob 文件名字为<filename>_shipped。在我们的例子中，blob 为 8123_bin.o_shipped，而且 kbuild 规则会从 8213_bin.o_shipped 创建一个副本并通过删去_shipped 部分创建文件 8123_bin.o。这样文件名 8123_bin.o 可用于模块的赋值中。

示例 4：

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

示例 4 所示，这与普通的.c/.c 文件和二进制文件没有什么不同。但是 kbuild 会采用不同的规则来创建.o 文件。

=== 5. 包含文件

当一个.c 文件使用其他.c 文件中的功能是需要包含文件（从 C 的意义上来说不是严格要求的，但是这是个好的编程习惯）。包含多于一个.c 文件的模块需要对应某个.c 文件构建一个.h 文件。

——如果.h 文件只描述模块内部接口，那么.h 文件需要与.c 文件放在相同目录下

——如果.h 文件描述了内核其他目录中用到的接口，那么.h 文件需要放到 include/linux/或者其他合适的 include/目录中

该规则的一个例外是在 include/下有自己的目录的大一点的子系统，例如 include/scsi。另一个例外是放在 include/asm-$(ARCH)/*目录中的特定体系结构的.h 文件

趋向于将外部模块的头文件放到单独的 include/目录下，这样需要在 kbuild 处理这种设置。

--- 5.1 如何从内核 include 目录中 include 文件

如果模块需要包含 include/linux/下的文件，那么使用：

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

kbuild 会确保向 gcc 添加选项以搜索相关目录。类似地，.h 文件和.c 文件下放在相同目录下时使用

```
#include "8123_if.h"
```

--- 5.2 在外部模块中使用 include/目录

外部模块通常将.h 文件放在单独的 include/目录下，虽然这不是常见的内核风格。如果外部模块使用了 include/目录，需要告诉 kbuild。这里可以使用的方法是使用 EXTRA_CFLAGS（对所有.c 文件有效）或 CFLAGS_$F.o（只对单个文件有效）。

在我们的例子中，如果将 8123_if.h 放到一个子目录 include/，那么 Kbuild 内容如下：

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

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

注意这里的赋值中-I 和路径之间没有空格。这是 kbuild 的限制：不能出现空格。

--- 5.3 在外部模块中使用多个目录

如果外部模块不采用常用的内核风格，而要将多个文件放于不同目录下，kbuild 也可以处理这种情况。

考虑下面的例子：

```
|
+- src/complex_main.c
|  +- hal/hardwareif.c
|  +- hal/include/hardwareif.h
+- include/complex.h
```

要构建模块 complex.ko，需要下面的 kbuild 文件：

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

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

kbuild 知道如何处理放在另一个目录下的.o 文件——虽然不推荐这种做法。使用的语法是指定对应 Kbuild 所在目录的相对目录。

要访问.h 文件，需要显示告诉 kbuild 到那里获取.h 文件。kbuild 执行的时候，当前目录总是内核树的根（参数-C），因此要告诉 kbuild 如何通过绝对路径获取.h 文件。$(src)指定了编译外部模块时 Kbuild 文件所在目录的绝对路径。因此-I$(src)/指出了 Kbuild 文件所在目录，其他目录附加到后面就可以。

=== 6. 安装模块

内核中包含的模块安装到目录：

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

外部模块安装到目录：

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

--- 6.1 INSTALL_MOD_PATH

上面是缺省目录，但是可以进行一些定制。可以通过变量 INSTALL_MOD_PATH 指定前缀：

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

INSTALL_MOD_PATH 设置为普通 shell 变量或者类似上面的例子在调用 make 时通过命令行指定。INSTALL_MOD_PATH 在安装内核内部模块或者外部模块时都有效。

--- 6.2 INSTALL_MOD_DIR

安装外部模块时缺省安装到/lib/modules/$(KERNELRELEASE)/extra，但是可能需要到单独的目录中访问模块以获取支持。为此，可以通过 INSTALL_MOD_DIR 指定不同于'extra'的目录：

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

=== 7. 模块版本化和 Module.symvers

通过 CONFIG_MODVERSIONS 开启模块版本化支持。

模块版本化用作简单的 ABI 移植性检测。模块版本化针对导出符号的完整原型计算一个 CRC 值，当加载/使用模块时会对比内核包含的 CRC 值和模块中的值。如果相等，那么内核拒绝加载模块。

Module.symvers 包含编译的内核中所有导出符号的列表。

--- 7.1 内核中的符号(vmlinux + modules)

编译内核时，会生成文件 Module.symvers。Module.symvers 包含内核和模块中的所有导出的符号；同时记录了每个符号对应的 CRC 值。

Module.symvers 文件的语法如下：

```
<CRC>     <Symbol>          <module>
```

示例：

```
0x2d036834  scsi_remove_host   drivers/scsi/scsi_mod
```

如果内核编译时不支持 CONFIG_MODVERSIONS，那么 crc 为 0x00000000
Module.symvers 有两个用途：

1) 列出 vmlinux 和所有模块中导出的符号

2) 如果激活 CONFIG_MODVERSIONS 则列出 CRC

--- 7.2 符号和外部模块

编译外部模块时，构建系统会访问内核中的符号以检测是否所有外部符号都定义。这在 MODPOST 阶段进行，并且 modpost 读取内核的 Module.symvers 文件以获取所有符号。

如果在编译外部模块的目录中有 Module.symvers 文件，那么也会读取该文件。在阶段 MODPOST 会构造一个新的 Module.symvers 文件，新文件中包含了内核中没有定义的所有导出符号。

--- 7.3 另一个外部模块中的符号

有时候，一个外部模块使用另一个外部模块中导出的符号。Kbuild 需要知道所有的符号以避免对未定义符号发出警告。有三种方法可以让 kbuild 知道多个外部模块的符号。推荐做法是使用顶层 kbuild 文件，但是在某些情况下该方法行不通。

使用顶层 Kbuild 文件

如果有两个模块'foo'和'bar'，而且'foo'使用'bar'中的符号，那么可以通过普通的顶层 kbuild 文件将两个模块同时加入编译内核中。

以如下目录布局为例：

```
./foo/ <= contains the foo module
./bar/ <= contains the bar module
```

顶层 Kbuild 文件中会如下编写：

```
#./Kbuild: (this file may also be named Makefile)
        obj-y := foo/ bar/
```

执行：

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

会执行预期操作，而且编译两个模块时会获取两个模块中的所有符号的信息。

使用额外的 Module.symvers 文件

编译外部模块时，会生成一个 Module.symvers 文件，该文件中包含内核中没有定义的导出符号。要访问'bar'中的符号，可以将编译'bar'模块时生成的 Module.symvers 文件复制到编译'foo'模块的目录中。编译模块时，kbuild 会读取外部模块目录中的 Module.symvers 文件，然后在编译结束时会生成一个新的 Module.symvers 文件，新文件中包含内核中没有定义的符号。

使用 Makefile 中的 make 变量 KBUILD_EXTRA_SYMBOLS

如果拷贝另一个模块的 Module.symvers 行不通，可以在 Makefile 中将 KBUILD_EXTRA_SYMBOLS 赋值为空格分隔的文件列表。在符号表初始化阶段 modpost 会加载这些文件。

=== 8. 技巧

--- 8.1 测试 CONFIG_FOO_BAR

模块有时需要检测一些 CONFIG_选项来决定是否要在模块中包含特定功能。使用 kbuild 时可以通过直接使用 CONFIG_变量做到这一点。

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

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

早期外部模块使用 grep 来直接检测.config 中特定的 CONFIG_设置，现在该方法不适用了。正如前面介绍的，编译外部模块时使用 kbuild，因此可以使用内核内部模块使用的测试 CONFIG_定义的方式。

【附录 A】00-INDEX

```
00-INDEX
        - this file: info on the kernel build process
kbuild.txt
        - developer information on kbuild
kconfig.txt
        - usage help for make *config
kconfig-language.txt
        - specification of Config Language, the language in Kconfig files
makefiles.txt
        - developer information for linux kernel makefiles
modules.txt
        - how to build modules and to install them
```

【附录 B】kbuild.txt

```
Environment variables

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.

KAFLAGS
---------------------------------------------------
Additional options to the assembler.

KCFLAGS
---------------------------------------------------
Additional options to the C compiler.

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.

KBUILD_EXTMOD
---------------------------------------------------
Set the directory to look for the kernel source when building external
modules.
The directory can be specified in several ways:
1) Use "M=..." on the command line
2) Environmnet variable KBUILD_EXTMOD
3) Environmnet variable SUBDIRS
The possibilities are listed in the order they take precedence.
Using "M=..." will always override the others.

KBUILD_OUTPUT
---------------------------------------------------
Specify the output directory when building the kernel.
```

```
37      The output directory can also be specificed using "O=...".
38      Setting "O=..." takes precedence over KBUILD_OUTPUT.
39
40      ARCH
41      ----------------------------------------------------
42      Set ARCH to the architecture to be built.
43      In most cases the name of the architecture is the same as the
44      directory name found in the arch/ directory.
45      But some architectures such as x86 and sparc have aliases.
46      x86: i386 for 32 bit, x86_64 for 64 bit
47      sparc: sparc for 32 bit, sparc64 for 64 bit
48
49      CROSS_COMPILE
50      ----------------------------------------------------
51      Specify an optional fixed part of the binutils filename.
52      CROSS_COMPILE can be a part of the filename or the full path.
53
54      CROSS_COMPILE is also used for ccache is some setups.
55
56      CF
57      ----------------------------------------------------
58      Additional options for sparse.
59      CF is often used on the command-line like this:
60
61          make CF=-Wbitwise C=2
62
63      INSTALL_PATH
64      ----------------------------------------------------
65      INSTALL_PATH specifies where to place the updated kernel and system map
66      images. Default is /boot, but you can set it to other values.
67
68
69      MODLIB
70      ----------------------------------------------------
71      Specify where to install modules.
72      The default value is:
73
74           $(INSTALL_MOD_PATH)/lib/modules/$(KERNELRELEASE)
75
76      The value can be overridden in which case the default value is ignored.
77
78      INSTALL_MOD_PATH
79      ----------------------------------------------------
80      INSTALL_MOD_PATH specifies a prefix to MODLIB for module directory
81      relocations required by build roots.  This is not defined in the
82      makefile but the argument can be passed to make if needed.
83
84      INSTALL_MOD_STRIP
85      ----------------------------------------------------
86      INSTALL_MOD_STRIP, if defined, will cause modules to be
87      stripped after they are installed.  If INSTALL_MOD_STRIP is '1', then
88      the default option --strip-debug will be used.  Otherwise,
89      INSTALL_MOD_STRIP will used as the options to the strip command.
```

```
90
91      INSTALL_FW_PATH
92      ----------------------------------------------------
93      INSTALL_FW_PATH specifies where to install the firmware blobs.
94      The default value is:
95
96          $(INSTALL_MOD_PATH)/lib/firmware
97
98      The value can be overridden in which case the default value is ignored.
99
100     INSTALL_HDR_PATH
101     ----------------------------------------------------
102     INSTALL_HDR_PATH specifies where to install user space headers when
103     executing "make headers_*".
104     The default value is:
105
106         $(objtree)/usr
107
108     $(objtree) is the directory where output files are saved.
109     The output directory is often set using "O=..." on the commandline.
110
111     The value can be overridden in which case the default value is ignored.
112
113     KBUILD_MODPOST_WARN
114     ----------------------------------------------------
115     KBUILD_MODPOST_WARN can be set to avoid errors in case of undefined
116     symbols in the final module linking stage. It changes such errors
117     into warnings.
118
119     KBUILD_MODPOST_NOFINAL
120     ----------------------------------------------------
121     KBUILD_MODPOST_NOFINAL can be set to skip the final link of modules.
122     This is solely useful to speed up test compiles.
123
124     KBUILD_EXTRA_SYMBOLS
125     ----------------------------------------------------
126     For modules that use symbols from other modules.
127     See more details in modules.txt.
128
129     ALLSOURCE_ARCHS
130     ----------------------------------------------------
131     For tags/TAGS/cscope targets, you can specify more than one arch
132     to be included in the databases, separated by blank space. E.g.:
133
134         $ make ALLSOURCE_ARCHS="x86 mips arm" tags
```

【附录 C】kconfig.txt

```
1       This file contains some assistance for using "make *config".
2
3       Use "make help" to list all of the possible configuration targets.
4
5       The xconfig ('qconf') and menuconfig ('mconf') programs also
6       have embedded help text.  Be sure to check it for navigation,
```

```
search, and other general help text.

======================================================================
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 when using "make oldconfig", use

       cp user/some/old.config .config
       yes "" | make oldconfig >conf.new

and the config program will list as (NEW) any new symbols that have
unknown values.  Of course, the .config file is also updated with
new (default) values, so you can use:

       grep "(NEW)" conf.new

to see the new config symbols or you can 'diff' the previous and
new .config files to see the differences:

       diff .config.old .config | less

(Yes, we need something better here.)

______________________________________________________________________
Environment variables for '*config'

KCONFIG_CONFIG
----------------------------------------------------
This environment variable can be used to specify a default kernel config
file name to override the default name of ".config".

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

KCONFIG_NOTIMESTAMP
----------------------------------------------------
If this environment variable exists and is non-null, the timestamp line
in generated .config files is omitted.

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

KCONFIG_ALLCONFIG
```

```
60      ---------------------------------------------------
61      (partially based on lkml email from/by Rob Landley, re: miniconfig)
62      ---------------------------------------------------
63      The allyesconfig/allmodconfig/allnoconfig/randconfig variants can
64      also use the environment variable KCONFIG_ALLCONFIG as a flag or a
65      filename that contains config symbols that the user requires to be
66      set to a specific value.  If KCONFIG_ALLCONFIG is used without a
67      filename, "make *config" checks for a file named
68      "all{yes/mod/no/random}.config" (corresponding to the *config command
69      that was used) for symbol values that are to be forced.  If this file
70      is not found, it checks for a file named "all.config" to contain forced
71      values.
72
73      This enables you to create "miniature" config (miniconfig) or custom
74      config files containing just the config symbols that you are interested
75      in.  Then the kernel config system generates the full .config file,
76      including symbols of your miniconfig file.
77
78      This 'KCONFIG_ALLCONFIG' file is a config file which contains
79      (usually a subset of all) preset config symbols.  These variable
80      settings are still subject to normal dependency checks.
81
82      Examples:
83              KCONFIG_ALLCONFIG=custom-notebook.config make allnoconfig
84      or
85              KCONFIG_ALLCONFIG=mini.config make allnoconfig
86      or
87              make KCONFIG_ALLCONFIG=mini.config allnoconfig
88
89      These examples will disable most options (allnoconfig) but enable or
90      disable the options that are explicitly listed in the specified
91      mini-config files.
92
93      ---------------------------------------------------------------------
94      Environment variables for 'silentoldconfig'
95
96      KCONFIG_NOSILENTUPDATE
97      ---------------------------------------------------
98      If this variable has a non-blank value, it prevents silent kernel
99      config udpates (requires explicit updates).
100
101     KCONFIG_AUTOCONFIG
102     ---------------------------------------------------
103     This environment variable can be set to specify the path & name of the
104     "auto.conf" file.  Its default value is "include/config/auto.conf".
105
106     KCONFIG_AUTOHEADER
107     ---------------------------------------------------
108     This environment variable can be set to specify the path & name of the
109     "autoconf.h" (header) file.  Its default value is "include/linux/autoconf.h".
110
111
112     =====================================================================
```

```
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, HOTPLUG_CPU, MEMORY_HOTPLUG.

       For search help, enter / followed TAB-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

_____________________________________________________________________
User interface options for 'menuconfig'

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)

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

Example:
       make MENUCONFIG_MODE=single_menu menuconfig


======================================================================
xconfig
--------------------------------------------------

Searching in xconfig:

```

```
166          The Search function searches for kernel configuration symbol
167          names, so you have to know something close to what you are
168          looking for.
169
170          Example:
171                 Ctrl-F hotplug
172          or
173                 Menu: File, Search, hotplug
174
175          lists all config symbol entries that contain "hotplug" in
176          the symbol name.  In this Search dialog, you may change the
177          config setting for any of the entries that are not grayed out.
178          You can also enter a different search string without having
179          to return to the main menu.
180
181
182    ========================================================================
183    gconfig
184    -------------------------------------------------
185
186    Searching in gconfig:
187
188          None (gconfig isn't maintained as well as xconfig or menuconfig);
189          however, gconfig does have a few more viewing choices than
190          xconfig does.
191
192    ###
```

【附录 D】kconfig-language.txt

```
1      Introduction
2      ------------
3
4      The configuration database is a collection of configuration options
5      organized in a tree structure:
6
7             +- Code maturity level options
8             |  +- Prompt for development and/or incomplete code/drivers
9             +- General setup
10            |  +- Networking support
11            |  +- System V IPC
12            |  +- BSD Process Accounting
13            |  +- Sysctl support
14            +- Loadable module support
15            |  +- Enable loadable module support
16            |     +- Set version information on all module symbols
17            |     +- Kernel module loader
18            +- ...
19
20     Every entry has its own dependencies. These dependencies are used
21     to determine the visibility of an entry. Any child entry is only
22     visible if its parent entry is also visible.
23
24     Menu entries
```

```
25    ------------
26
27    Most entries define a config option; all other entries help to organize
28    them. A single configuration option is defined like this:
29
30    config MODVERSIONS
31           bool "Set version information on all module symbols"
32           depends on MODULES
33           help
34             Usually, modules have to be recompiled whenever you switch to a new
35             kernel.  ...
36
37    Every line starts with a key word and can be followed by multiple
38    arguments.  "config" starts a new config entry. The following lines
39    define attributes for this config option. Attributes can be the type of
40    the config option, input prompt, dependencies, help text and default
41    values. A config option can be defined multiple times with the same
42    name, but every definition can have only a single input prompt and the
43    type must not conflict.
44
45    Menu attributes
46    ---------------
47
48    A menu entry can have a number of attributes. Not all of them are
49    applicable everywhere (see syntax).
50
51    - type definition: "bool"/"tristate"/"string"/"hex"/"int"
52      Every config option must have a type. There are only two basic types:
53      tristate and string; the other types are based on these two. The type
54      definition optionally accepts an input prompt, so these two examples
55      are equivalent:
56
57           bool "Networking support"
58      and
59           bool
60           prompt "Networking support"
61
62    - input prompt: "prompt" <prompt> ["if" <expr>]
63      Every menu entry can have at most one prompt, which is used to display
64      to the user. Optionally dependencies only for this prompt can be added
65      with "if".
66
67    - default value: "default" <expr> ["if" <expr>]
68      A config option can have any number of default values. If multiple
69      default values are visible, only the first defined one is active.
70      Default values are not limited to the menu entry where they are
71      defined. This means the default can be defined somewhere else or be
72      overridden by an earlier definition.
73      The default value is only assigned to the config symbol if no other
74      value was set by the user (via the input prompt above). If an input
75      prompt is visible the default value is presented to the user and can
76      be overridden by him.
77      Optionally, dependencies only for this default value can be added with
```

```
78        "if".
79
80      - type definition + default value:
81             "def_bool"/"def_tristate" <expr> ["if" <expr>]
82        This is a shorthand notation for a type definition plus a value.
83        Optionally dependencies for this default value can be added with "if".
84
85      - dependencies: "depends on" <expr>
86        This defines a dependency for this menu entry. If multiple
87        dependencies are defined, they are connected with '&&'. Dependencies
88        are applied to all other options within this menu entry (which also
89        accept an "if" expression), so these two examples are equivalent:
90
91             bool "foo" if BAR
92             default y if BAR
93        and
94             depends on BAR
95             bool "foo"
96             default y
97
98      - reverse dependencies: "select" <symbol> ["if" <expr>]
99        While normal dependencies reduce the upper limit of a symbol (see
100       below), reverse dependencies can be used to force a lower limit of
101       another symbol. The value of the current menu symbol is used as the
102       minimal value <symbol> can be set to. If <symbol> is selected multiple
103       times, the limit is set to the largest selection.
104       Reverse dependencies can only be used with boolean or tristate
105       symbols.
106       Note:
107            select should be used with care. select will force
108            a symbol to a value without visiting the dependencies.
109            By abusing select you are able to select a symbol FOO even
110            if FOO depends on BAR that is not set.
111            In general use select only for non-visible symbols
112            (no prompts anywhere) and for symbols with no dependencies.
113            That will limit the usefulness but on the other hand avoid
114            the illegal configurations all over.
115            kconfig should one day warn about such things.
116
117     - numerical ranges: "range" <symbol> <symbol> ["if" <expr>]
118       This allows to limit the range of possible input values for int
119       and hex symbols. The user can only input a value which is larger than
120       or equal to the first symbol and smaller than or equal to the second
121       symbol.
122
123     - help text: "help" or "---help---"
124       This defines a help text. The end of the help text is determined by
125       the indentation level, this means it ends at the first line which has
126       a smaller indentation than the first line of the help text.
127       "---help---" and "help" do not differ in behaviour, "---help---" is
128       used to help visually separate configuration logic from help within
129       the file as an aid to developers.
130
```

- 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

```
184  respectively for calculations). A menu entry becomes visible when it's
185  expression evaluates to 'm' or 'y'.
186
187  There are two types of symbols: constant and non-constant symbols.
188  Non-constant symbols are the most common ones and are defined with the
189  'config' statement. Non-constant symbols consist entirely of alphanumeric
190  characters or underscores.
191  Constant symbols are only part of expressions. Constant symbols are
192  always surrounded by single or double quotes. Within the quote, any
193  other character is allowed and the quotes can be escaped using '\'.
194
195  Menu structure
196  --------------
197
198  The position of a menu entry in the tree is determined in two ways. First
199  it can be specified explicitly:
200
201  menu "Network device support"
202          depends on NET
203
204  config NETDEVICES
205          ...
206
207  endmenu
208
209  All entries within the "menu" ... "endmenu" block become a submenu of
210  "Network device support". All subentries inherit the dependencies from
211  the menu entry, e.g. this means the dependency "NET" is added to the
212  dependency list of the config option NETDEVICES.
213
214  The other way to generate the menu structure is done by analyzing the
215  dependencies. If a menu entry somehow depends on the previous entry, it
216  can be made a submenu of it. First, the previous (parent) symbol must
217  be part of the dependency list and then one of these two conditions
218  must be true:
219  - the child entry must become invisible, if the parent is set to 'n'
220  - the child entry must only be visible, if the parent is visible
221
222  config MODULES
223          bool "Enable loadable module support"
224
225  config MODVERSIONS
226          bool "Set version information on all module symbols"
227          depends on MODULES
228
229  comment "module support disabled"
230          depends on !MODULES
231
232  MODVERSIONS directly depends on MODULES, this means it's only visible if
233  MODULES is different from 'n'. The comment on the other hand is always
234  visible when MODULES is visible (the (empty) dependency of MODULES is
235  also part of the comment dependencies).
236
```

```
237
238   Kconfig syntax
239   --------------
240
241   The configuration file describes a series of menu entries, where every
242   line starts with a keyword (except help texts). The following keywords
243   end a menu entry:
244   - config
245   - menuconfig
246   - choice/endchoice
247   - comment
248   - menu/endmenu
249   - if/endif
250   - source
251   The first five also start the definition of a menu entry.
252
253   config:
254
255          "config" <symbol>
256          <config options>
257
258   This defines a config symbol <symbol> and accepts any of above
259   attributes as options.
260
261   menuconfig:
262          "menuconfig" <symbol>
263          <config options>
264
265   This is similar to the simple config entry above, but it also gives a
266   hint to front ends, that all suboptions should be displayed as a
267   separate list of options.
268
269   choices:
270
271          "choice"
272          <choice options>
273          <choice block>
274          "endchoice"
275
276   This defines a choice group and accepts any of the above attributes as
277   options. A choice can only be of type bool or tristate, while a boolean
278   choice only allows a single config entry to be selected, a tristate
279   choice also allows any number of config entries to be set to 'm'. This
280   can be used if multiple drivers for a single hardware exists and only a
281   single driver can be compiled/loaded into the kernel, but all drivers
282   can be compiled as modules.
283   A choice accepts another option "optional", which allows to set the
284   choice to 'n' and no entry needs to be selected.
285
286   comment:
287
288          "comment" <prompt>
289          <comment options>
```

290
291 This defines a comment which is displayed to the user during the
292 configuration process and is also echoed to the output files. The only
293 possible options are dependencies.
294
295 menu:
296
297 "menu" <prompt>
298 <menu options>
299 <menu block>
300 "endmenu"
301
302 This defines a menu block, see "Menu structure" above for more
303 information. The only possible options are dependencies.
304
305 if:
306
307 "if" <expr>
308 <if block>
309 "endif"
310
311 This defines an if block. The dependency expression <expr> is appended
312 to all enclosed menu entries.
313
314 source:
315
316 "source" <prompt>
317
318 This reads the specified configuration file. This file is always parsed.
319
320 mainmenu:
321
322 "mainmenu" <prompt>
323
324 This sets the config program's title bar if the config program chooses
325 to use it.
326
327
328 Kconfig hints
329 -------------
330 This is a collection of Kconfig tips, most of which aren't obvious at
331 first glance and most of which have become idioms in several Kconfig
332 files.
333
334 Adding common features and make the usage configurable
335 ~~~~~~
336 It is a common idiom to implement a feature/functionality that are
337 relevant for some architectures but not all.
338 The recommended way to do so is to use a config variable named HAVE_*
339 that is defined in a common Kconfig file and selected by the relevant
340 architectures.
341 An example is the generic IOMAP functionality.
342

```
343    We would in lib/Kconfig see:
344
345    # Generic IOMAP is used to ...
346    config HAVE_GENERIC_IOMAP
347
348    config GENERIC_IOMAP
349           depends on HAVE_GENERIC_IOMAP && FOO
350
351    And in lib/Makefile we would see:
352    obj-$(CONFIG_GENERIC_IOMAP) += iomap.o
353
354    For each architecture using the generic IOMAP functionality we would see:
355
356    config X86
357           select ...
358           select HAVE_GENERIC_IOMAP
359           select ...
360
361    Note: we use the existing config option and avoid creating a new
362    config variable to select HAVE_GENERIC_IOMAP.
363
364    Note: the use of the internal config variable HAVE_GENERIC_IOMAP, it is
365    introduced to overcome the limitation of select which will force a
366    config option to 'y' no matter the dependencies.
367    The dependencies are moved to the symbol GENERIC_IOMAP and we avoid the
368    situation where select forces a symbol equals to 'y'.
369
370    Build as module only
371    ~~~~~~~~~~~~~~~~~~~~
372    To restrict a component build to module-only, qualify its config symbol
373    with "depends on m".  E.g.:
374
375    config FOO
376           depends on BAR && m
377
378    limits FOO to module (=m) or disabled (=n).
379
```

【附录 E】makefiles.txt

```
1      Linux Kernel Makefiles
2
3      This document describes the Linux kernel Makefiles.
4
5      === Table of Contents
6
7             === 1 Overview
8             === 2 Who does what
9             === 3 The kbuild files
10               --- 3.1 Goal definitions
11               --- 3.2 Built-in object goals - obj-y
12               --- 3.3 Loadable module goals - obj-m
13               --- 3.4 Objects which export symbols
14               --- 3.5 Library file goals - lib-y
```

```
15              --- 3.6 Descending down in directories
16              --- 3.7 Compilation flags
17              --- 3.8 Command line dependency
18              --- 3.9 Dependency tracking
19              --- 3.10 Special Rules
20              --- 3.11 $(CC) support functions
21
22           === 4 Host Program support
23              --- 4.1 Simple Host Program
24              --- 4.2 Composite Host Programs
25              --- 4.3 Defining shared libraries
26              --- 4.4 Using C++ for host programs
27              --- 4.5 Controlling compiler options for host programs
28              --- 4.6 When host programs are actually built
29              --- 4.7 Using hostprogs-$(CONFIG_FOO)
30
31           === 5 Kbuild clean infrastructure
32
33           === 6 Architecture Makefiles
34              --- 6.1 Set variables to tweak the build to the architecture
35              --- 6.2 Add prerequisites to archprepare:
36              --- 6.3 List directories to visit when descending
37              --- 6.4 Architecture-specific boot images
38              --- 6.5 Building non-kbuild targets
39              --- 6.6 Commands useful for building a boot image
40              --- 6.7 Custom kbuild commands
41              --- 6.8 Preprocessing linker scripts
42
43           === 7 Kbuild syntax for exported headers
44                  --- 7.1 header-y
45                  --- 7.2 objhdr-y
46                  --- 7.3 destination-y
47                  --- 7.4 unifdef-y (deprecated)
48
49           === 8 Kbuild Variables
50           === 9 Makefile language
51           === 10 Credits
52           === 11 TODO
53
54     === 1 Overview
55
56     The Makefiles have five parts:
57
58           Makefile             the top Makefile.
59           .config              the kernel configuration file.
60           arch/$(ARCH)/Makefile the arch Makefile.
61           scripts/Makefile.*   common rules etc. for all kbuild Makefiles.
62           kbuild Makefiles     there are about 500 of these.
63
64     The top Makefile reads the .config file, which comes from the kernel
65     configuration process.
66
67     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/$(ARCH)/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.

=== 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.

=== 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 3.1 "Goal definitions" is a quick intro, further chapters provide
more details, with real examples.

```
121
122    --- 3.1 Goal definitions
123
124            Goal definitions are the main part (heart) of the kbuild Makefile.
125            These lines define the files to be built, any special compilation
126            options, and any subdirectories to be entered recursively.
127
128            The most simple kbuild makefile contains one line:
129
130            Example:
131                    obj-y += foo.o
132
133            This tells kbuild that there is one object in that directory, named
134            foo.o.  foo.o will be built from foo.c or foo.S.
135
136            If foo.o shall be built as a module, the variable obj-m is used.
137            Therefore the following pattern is often used:
138
139            Example:
140                    obj-$(CONFIG_FOO) += foo.o
141
142            $(CONFIG_FOO) evaluates to either y (for built-in) or m (for module).
143            If CONFIG_FOO is neither y nor m, then the file will not be compiled
144            nor linked.
145
146    --- 3.2 Built-in object goals - obj-y
147
148            The kbuild Makefile specifies object files for vmlinux
149            in the $(obj-y) lists.  These lists depend on the kernel
150            configuration.
151
152            Kbuild compiles all the $(obj-y) files.  It then calls
153            "$(LD) -r" to merge these files into one built-in.o file.
154            built-in.o is later linked into vmlinux by the parent Makefile.
155
156            The order of files in $(obj-y) is significant.  Duplicates in
157            the lists are allowed: the first instance will be linked into
158            built-in.o and succeeding instances will be ignored.
159
160            Link order is significant, because certain functions
161            (module_init() / __initcall) will be called during boot in the
162            order they appear. So keep in mind that changing the link
163            order may e.g. change the order in which your SCSI
164            controllers are detected, and thus your disks are renumbered.
165
166            Example:
167                    #drivers/isdn/i4l/Makefile
168                    # Makefile for the kernel ISDN subsystem and device drivers.
169                    # Each configuration option enables a list of files.
170                    obj-$(CONFIG_ISDN)              += isdn.o
171                    obj-$(CONFIG_ISDN_PPP_BSDCOMP) += isdn_bsdcomp.o
172
173    --- 3.3 Loadable module goals - obj-m
```

```
174
175         $(obj-m) specify object files which are built as loadable
176         kernel modules.
177
178         A module may be built from one source file or several source
179         files. In the case of one source file, the kbuild makefile
180         simply adds the file to $(obj-m).
181
182         Example:
183                #drivers/isdn/i4l/Makefile
184                obj-$(CONFIG_ISDN_PPP_BSDCOMP) += isdn_bsdcomp.o
185
186         Note: In this example $(CONFIG_ISDN_PPP_BSDCOMP) evaluates to 'm'
187
188         If a kernel module is built from several source files, you specify
189         that you want to build a module in the same way as above.
190
191         Kbuild needs to know which the parts that you want to build your
192         module from, so you have to tell it by setting an
193         $(<module_name>-objs) variable.
194
195         Example:
196                #drivers/isdn/i4l/Makefile
197                obj-$(CONFIG_ISDN) += isdn.o
198                isdn-objs := isdn_net_lib.o isdn_v110.o isdn_common.o
199
200         In this example, the module name will be isdn.o. Kbuild will
201         compile the objects listed in $(isdn-objs) and then run
202         "$(LD) -r" on the list of these files to generate isdn.o.
203
204         Kbuild recognises objects used for composite objects by the suffix
205         -objs, and the suffix -y. This allows the Makefiles to use
206         the value of a CONFIG_ symbol to determine if an object is part
207         of a composite object.
208
209         Example:
210                #fs/ext2/Makefile
211                 obj-$(CONFIG_EXT2_FS)        += ext2.o
212                ext2-y                       := balloc.o bitmap.o
213                 ext2-$(CONFIG_EXT2_FS_XATTR) += xattr.o
214
215         In this example, xattr.o is only part of the composite object
216         ext2.o if $(CONFIG_EXT2_FS_XATTR) evaluates to 'y'.
217
218         Note: Of course, when you are building objects into the kernel,
219         the syntax above will also work. So, if you have CONFIG_EXT2_FS=y,
220         kbuild will build an ext2.o file for you out of the individual
221         parts and then link this into built-in.o, as you would expect.
222
223    --- 3.4 Objects which export symbols
224
225         No special notation is required in the makefiles for
226         modules exporting symbols.
```

```
227
228     --- 3.5 Library file goals - lib-y
229
230             Objects listed with obj-* are used for modules, or
231             combined in a built-in.o for that specific directory.
232             There is also the possibility to list objects that will
233             be included in a library, lib.a.
234             All objects listed with lib-y are combined in a single
235             library for that directory.
236             Objects that are listed in obj-y and additionally listed in
237             lib-y will not be included in the library, since they will
238             be accessible anyway.
239             For consistency, objects listed in lib-m will be included in lib.a.
240
241             Note that the same kbuild makefile may list files to be built-in
242             and to be part of a library. Therefore the same directory
243             may contain both a built-in.o and a lib.a file.
244
245             Example:
246                     #arch/i386/lib/Makefile
247                     lib-y    := checksum.o delay.o
248
249             This will create a library lib.a based on checksum.o and delay.o.
250             For kbuild to actually recognize that there is a lib.a being built,
251             the directory shall be listed in libs-y.
252             See also "6.3 List directories to visit when descending".
253
254             Use of lib-y is normally restricted to lib/ and arch/*/lib.
255
256     --- 3.6 Descending down in directories
257
258             A Makefile is only responsible for building objects in its own
259             directory. Files in subdirectories should be taken care of by
260             Makefiles in these subdirs. The build system will automatically
261             invoke make recursively in subdirectories, provided you let it know of
262             them.
263
264             To do so, obj-y and obj-m are used.
265             ext2 lives in a separate directory, and the Makefile present in fs/
266             tells kbuild to descend down using the following assignment.
267
268             Example:
269                     #fs/Makefile
270                     obj-$(CONFIG_EXT2_FS) += ext2/
271
272             If CONFIG_EXT2_FS is set to either 'y' (built-in) or 'm' (modular)
273             the corresponding obj- variable will be set, and kbuild will descend
274             down in the ext2 directory.
275             Kbuild only uses this information to decide that it needs to visit
276             the directory, it is the Makefile in the subdirectory that
277             specifies what is modules and what is built-in.
278
279             It is good practice to use a CONFIG_ variable when assigning directory
```

```
280          names. This allows kbuild to totally skip the directory if the
281          corresponding CONFIG_ option is neither 'y' nor 'm'.
282
283    --- 3.7 Compilation flags
284
285        ccflags-y, asflags-y and ldflags-y
286          The three flags listed above applies only to the kbuild makefile
287          where they are assigned. They are used for all the normal
288          cc, as and ld invocation happenign during a recursive build.
289          Note: Flags with the same behaviour were previously named:
290          EXTRA_CFLAGS, EXTRA_AFLAGS and EXTRA_LDFLAGS.
291          They are yet supported but their use are deprecated.
292
293          ccflags-y specifies options for compiling C files with $(CC).
294
295          Example:
296                 # drivers/sound/emu10k1/Makefile
297                 ccflags-y += -I$(obj)
298                 ccflags-$(DEBUG) += -DEMU10K1_DEBUG
299
300
301          This variable is necessary because the top Makefile owns the
302          variable $(KBUILD_CFLAGS) and uses it for compilation flags for the
303          entire tree.
304
305          asflags-y is a similar string for per-directory options
306          when compiling assembly language source.
307
308          Example:
309                 #arch/x86_64/kernel/Makefile
310                 asflags-y := -traditional
311
312
313          ldflags-y is a string for per-directory options to $(LD).
314
315          Example:
316                 #arch/m68k/fpsp040/Makefile
317                 ldflags-y := -x
318
319        subdir-ccflags-y, subdir-asflags-y
320          The two flags listed above are similar to ccflags-y and as-falgs-y.
321          The difference is that the subdir- variants has effect for the kbuild
322          file where tey are present and all subdirectories.
323          Options specified using subdir-* are added to the commandline before
324          the options specified using the non-subdir variants.
325
326          Example:
327                 subdir-ccflags-y := -Werror
328
329        CFLAGS_$@, AFLAGS_$@
330
331          CFLAGS_$@ and AFLAGS_$@ only apply to commands in current
332          kbuild makefile.
```

```
333
334          $(CFLAGS_$@) specifies per-file options for $(CC).  The $@
335          part has a literal value which specifies the file that it is for.
336
337          Example:
338                 # drivers/scsi/Makefile
339                 CFLAGS_aha152x.o =   -DAHA152X_STAT -DAUTOCONF
340                 CFLAGS_gdth.o    = # -DDEBUG_GDTH=2 -D__SERIAL__ -D__COM2__ \
341                                      -DGDTH_STATISTICS
342                 CFLAGS_seagate.o =   -DARBITRATE -DPARITY -DSEAGATE_USE_ASM
343
344          These three lines specify compilation flags for aha152x.o,
345          gdth.o, and seagate.o
346
347          $(AFLAGS_$@) is a similar feature for source files in assembly
348          languages.
349
350          Example:
351                 # arch/arm/kernel/Makefile
352                 AFLAGS_head-armv.o := -DTEXTADDR=$(TEXTADDR) -traditional
353                 AFLAGS_head-armo.o := -DTEXTADDR=$(TEXTADDR) -traditional
354
355    --- 3.9 Dependency tracking
356
357          Kbuild tracks dependencies on the following:
358          1) All prerequisite files (both *.c and *.h)
359          2) CONFIG_ options used in all prerequisite files
360          3) Command-line used to compile target
361
362          Thus, if you change an option to $(CC) all affected files will
363          be re-compiled.
364
365    --- 3.10 Special Rules
366
367          Special rules are used when the kbuild infrastructure does
368          not provide the required support. A typical example is
369          header files generated during the build process.
370          Another example are the architecture-specific Makefiles which
371          need special rules to prepare boot images etc.
372
373          Special rules are written as normal Make rules.
374          Kbuild is not executing in the directory where the Makefile is
375          located, so all special rules shall provide a relative
376          path to prerequisite files and target files.
377
378          Two variables are used when defining special rules:
379
380        $(src)
381          $(src) is a relative path which points to the directory
382          where the Makefile is located. Always use $(src) when
383          referring to files located in the src tree.
384
385        $(obj)
```

```
386          $(obj) is a relative path which points to the directory
387          where the target is saved. Always use $(obj) when
388          referring to generated files.
389
390          Example:
391                 #drivers/scsi/Makefile
392                 $(obj)/53c8xx_d.h: $(src)/53c7,8xx.scr $(src)/script_asm.pl
393                         $(CPP) -DCHIP=810 - < $< | ... $(src)/script_asm.pl
394
395          This is a special rule, following the normal syntax
396          required by make.
397          The target file depends on two prerequisite files. References
398          to the target file are prefixed with $(obj), references
399          to prerequisites are referenced with $(src) (because they are not
400          generated files).
401
402       $(kecho)
403          echoing information to user in a rule is often a good practice
404          but when execution "make -s" one does not expect to see any output
405          except for warnings/errors.
406          To support this kbuild define $(kecho) which will echo out the
407          text following $(kecho) to stdout except if "make -s" is used.
408
409          Example:
410                 #arch/blackfin/boot/Makefile
411                 $(obj)/vmImage: $(obj)/vmlinux.gz
412                         $(call if_changed,uimage)
413                         @$(kecho) 'Kernel: $@ is ready'
414
415
416    --- 3.11 $(CC) support functions
417
418          The kernel may be built with several different versions of
419          $(CC), each supporting a unique set of features and options.
420          kbuild provide basic support to check for valid options for $(CC).
421          $(CC) is usually the gcc compiler, but other alternatives are
422          available.
423
424       as-option
425          as-option is used to check if $(CC) -- when used to compile
426          assembler (*.S) files -- supports the given option. An optional
427          second option may be specified if the first option is not supported.
428
429          Example:
430                 #arch/sh/Makefile
431                 cflags-y += $(call as-option,-Wa$(comma)-isa=$(isa-y),)
432
433          In the above example, cflags-y will be assigned the option
434          -Wa$(comma)-isa=$(isa-y) if it is supported by $(CC).
435          The second argument is optional, and if supplied will be used
436          if first argument is not supported.
437
438       ld-option
```

```
439          ld-option is used to check if $(CC) when used to link object files
440          supports the given option.  An optional second option may be
441          specified if first option are not supported.
442
443          Example:
444                 #arch/i386/kernel/Makefile
445                 vsyscall-flags += $(call ld-option, -Wl$(comma)--hash-style=sysv)
446
447          In the above example, vsyscall-flags will be assigned the option
448          -Wl$(comma)--hash-style=sysv if it is supported by $(CC).
449          The second argument is optional, and if supplied will be used
450          if first argument is not supported.
451
452       as-instr
453          as-instr checks if the assembler reports a specific instruction
454          and then outputs either option1 or option2
455          C escapes are supported in the test instruction
456          Note: as-instr-option uses KBUILD_AFLAGS for $(AS) options
457
458       cc-option
459          cc-option is used to check if $(CC) supports a given option, and not
460          supported to use an optional second option.
461
462          Example:
463                 #arch/i386/Makefile
464                 cflags-y += $(call cc-option,-march=pentium-mmx,-march=i586)
465
466          In the above example, cflags-y will be assigned the option
467          -march=pentium-mmx if supported by $(CC), otherwise -march=i586.
468          The second argument to cc-option is optional, and if omitted,
469          cflags-y will be assigned no value if first option is not supported.
470          Note: cc-option uses KBUILD_CFLAGS for $(CC) options
471
472       cc-option-yn
473          cc-option-yn is used to check if gcc supports a given option
474          and return 'y' if supported, otherwise 'n'.
475
476          Example:
477                 #arch/ppc/Makefile
478                 biarch := $(call cc-option-yn, -m32)
479                 aflags-$(biarch) += -a32
480                 cflags-$(biarch) += -m32
481
482          In the above example, $(biarch) is set to y if $(CC) supports the -m32
483          option. When $(biarch) equals 'y', the expanded variables $(aflags-y)
484          and $(cflags-y) will be assigned the values -a32 and -m32,
485          respectively.
486          Note: cc-option-yn uses KBUILD_CFLAGS for $(CC) options
487
488       cc-option-align
489          gcc versions >= 3.0 changed the type of options used to specify
490          alignment of functions, loops etc. $(cc-option-align), when used
491          as prefix to the align options, will select the right prefix:
```

```
492          gcc < 3.00
493                 cc-option-align = -malign
494          gcc >= 3.00
495                 cc-option-align = -falign
496
497          Example:
498                 KBUILD_CFLAGS += $(cc-option-align)-functions=4
499
500          In the above example, the option -falign-functions=4 is used for
501          gcc >= 3.00. For gcc < 3.00, -malign-functions=4 is used.
502          Note: cc-option-align uses KBUILD_CFLAGS for $(CC) options
503
504       cc-version
505          cc-version returns a numerical version of the $(CC) compiler version.
506          The format is <major><minor> where both are two digits. So for example
507          gcc 3.41 would return 0341.
508          cc-version is useful when a specific $(CC) version is faulty in one
509          area, for example -mregparm=3 was broken in some gcc versions
510          even though the option was accepted by gcc.
511
512          Example:
513                 #arch/i386/Makefile
514                 cflags-y += $(shell \
515                 if [ $(call cc-version) -ge 0300 ] ; then \
516                        echo "-mregparm=3"; fi ;)
517
518          In the above example, -mregparm=3 is only used for gcc version greater
519          than or equal to gcc 3.0.
520
521       cc-ifversion
522          cc-ifversion tests the version of $(CC) and equals last argument if
523          version expression is true.
524
525          Example:
526                 #fs/reiserfs/Makefile
527                 ccflags-y := $(call cc-ifversion, -lt, 0402, -O1)
528
529          In this example, ccflags-y will be assigned the value -O1 if the
530          $(CC) version is less than 4.2.
531          cc-ifversion takes all the shell operators:
532          -eq, -ne, -lt, -le, -gt, and -ge
533          The third parameter may be a text as in this example, but it may also
534          be an expanded variable or a macro.
535
536       cc-fullversion
537          cc-fullversion is useful when the exact version of gcc is needed.
538          One typical use-case is when a specific GCC version is broken.
539          cc-fullversion points out a more specific version than cc-version does.
540
541          Example:
542                 #arch/powerpc/Makefile
543                 $(Q)if test "$(call cc-fullversion)" = "040200" ; then \
544                        echo -n '*** GCC-4.2.0 cannot compile the 64-bit powerpc ' ; \
```

```
                        false ; \
                fi

        In this example for a specific GCC version the build will error out explaining
        to the user why it stops.

    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

=== 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-y.

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).
Both possibilities are described in the following.

--- 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-y := bin2hex

        Kbuild assumes in the above example that bin2hex is made from a single
```

```
598          c-source file named bin2hex.c located in the same directory as
599          the Makefile.
600
601    --- 4.2 Composite Host Programs
602
603          Host programs can be made up based on composite objects.
604          The syntax used to define composite objects for host programs is
605          similar to the syntax used for kernel objects.
606          $(<executable>-objs) lists all objects used to link the final
607          executable.
608
609          Example:
610                #scripts/lxdialog/Makefile
611                hostprogs-y   := lxdialog
612                lxdialog-objs := checklist.o lxdialog.o
613
614          Objects with extension .o are compiled from the corresponding .c
615          files. In the above example, checklist.c is compiled to checklist.o
616          and lxdialog.c is compiled to lxdialog.o.
617          Finally, the two .o files are linked to the executable, lxdialog.
618          Note: The syntax <executable>-y is not permitted for host-programs.
619
620    --- 4.3 Defining shared libraries
621
622          Objects with extension .so are considered shared libraries, and
623          will be compiled as position independent objects.
624          Kbuild provides support for shared libraries, but the usage
625          shall be restricted.
626          In the following example the libkconfig.so shared library is used
627          to link the executable conf.
628
629          Example:
630                #scripts/kconfig/Makefile
631                hostprogs-y     := conf
632                conf-objs       := conf.o libkconfig.so
633                libkconfig-objs := expr.o type.o
634
635          Shared libraries always require a corresponding -objs line, and
636          in the example above the shared library libkconfig is composed by
637          the two objects expr.o and type.o.
638          expr.o and type.o will be built as position independent code and
639          linked as a shared library libkconfig.so. C++ is not supported for
640          shared libraries.
641
642    --- 4.4 Using C++ for host programs
643
644          kbuild offers support for host programs written in C++. This was
645          introduced solely to support kconfig, and is not recommended
646          for general use.
647
648          Example:
649                #scripts/kconfig/Makefile
650                hostprogs-y   := qconf
```

```
651                qconf-cxxobjs := qconf.o
652
653          In the example above the executable is composed of the C++ file
654          qconf.cc - identified by $(qconf-cxxobjs).
655
656          If qconf is composed by a mixture of .c and .cc files, then an
657          additional line can be used to identify this.
658
659          Example:
660                #scripts/kconfig/Makefile
661                hostprogs-y   := qconf
662                qconf-cxxobjs := qconf.o
663                qconf-objs    := check.o
664
665    --- 4.5 Controlling compiler options for host programs
666
667          When compiling host programs, it is possible to set specific flags.
668          The programs will always be compiled utilising $(HOSTCC) passed
669          the options specified in $(HOSTCFLAGS).
670          To set flags that will take effect for all host programs created
671          in that Makefile, use the variable HOST_EXTRACFLAGS.
672
673          Example:
674                #scripts/lxdialog/Makefile
675                HOST_EXTRACFLAGS += -I/usr/include/ncurses
676
677          To set specific flags for a single file the following construction
678          is used:
679
680          Example:
681                #arch/ppc64/boot/Makefile
682                HOSTCFLAGS_piggyback.o := -DKERNELBASE=$(KERNELBASE)
683
684          It is also possible to specify additional options to the linker.
685
686          Example:
687                #scripts/kconfig/Makefile
688                HOSTLOADLIBES_qconf := -L$(QTDIR)/lib
689
690          When linking qconf, it will be passed the extra option
691          "-L$(QTDIR)/lib".
692
693    --- 4.6 When host programs are actually built
694
695          Kbuild will only build host-programs when they are referenced
696          as a prerequisite.
697          This is possible in two ways:
698
699          (1) List the prerequisite explicitly in a special rule.
700
701          Example:
702                #drivers/pci/Makefile
703                hostprogs-y := gen-devlist
```

```
704                 $(obj)/devlist.h: $(src)/pci.ids $(obj)/gen-devlist
705                         ( cd $(obj); ./gen-devlist ) < $<
706
707         The target $(obj)/devlist.h will not be built before
708         $(obj)/gen-devlist is updated. Note that references to
709         the host programs in special rules must be prefixed with $(obj).
710
711         (2) Use $(always)
712         When there is no suitable special rule, and the host program
713         shall be built when a makefile is entered, the $(always)
714         variable shall be used.
715
716         Example:
717                 #scripts/lxdialog/Makefile
718                 hostprogs-y   := lxdialog
719                 always        := $(hostprogs-y)
720
721         This will tell kbuild to build lxdialog even if not referenced in
722         any rule.
723
724 --- 4.7 Using hostprogs-$(CONFIG_FOO)
725
726         A typical pattern in a Kbuild file looks like this:
727
728         Example:
729                 #scripts/Makefile
730                 hostprogs-$(CONFIG_KALLSYMS) += kallsyms
731
732         Kbuild knows about both 'y' for built-in and 'm' for module.
733         So if a config symbol evaluate to 'm', kbuild will still build
734         the binary. In other words, Kbuild handles hostprogs-m exactly
735         like hostprogs-y. But only hostprogs-y is recommended to be used
736         when no CONFIG symbols are involved.
737
738 === 5 Kbuild clean infrastructure
739
740 "make clean" deletes most generated files in the obj tree where the kernel
741 is compiled. This includes generated files such as host programs.
742 Kbuild knows targets listed in $(hostprogs-y), $(hostprogs-m), $(always),
743 $(extra-y) and $(targets). They are all deleted during "make clean".
744 Files matching the patterns "*.[oas]", "*.ko", plus some additional files
745 generated by kbuild are deleted all over the kernel src tree when
746 "make clean" is executed.
747
748 Additional files can be specified in kbuild makefiles by use of $(clean-files).
749
750         Example:
751                 #drivers/pci/Makefile
752                 clean-files := devlist.h classlist.h
753
754 When executing "make clean", the two files "devlist.h classlist.h" will
755 be deleted. Kbuild will assume files to be in same relative directory as the
756 Makefile except if an absolute path is specified (path starting with '/').
```

```
757
758   To delete a directory hierarchy use:
759
760           Example:
761                   #scripts/package/Makefile
762                   clean-dirs := $(objtree)/debian/
763
764   This will delete the directory debian, including all subdirectories.
765   Kbuild will assume the directories to be in the same relative path as the
766   Makefile if no absolute path is specified (path does not start with '/').
767
768   Usually kbuild descends down in subdirectories due to "obj-* := dir/",
769   but in the architecture makefiles where the kbuild infrastructure
770   is not sufficient this sometimes needs to be explicit.
771
772           Example:
773                   #arch/i386/boot/Makefile
774                   subdir- := compressed/
775
776   The above assignment instructs kbuild to descend down in the
777   directory compressed/ when "make clean" is executed.
778
779   To support the clean infrastructure in the Makefiles that builds the
780   final bootimage there is an optional target named archclean:
781
782           Example:
783                   #arch/i386/Makefile
784                   archclean:
785                           $(Q)$(MAKE) $(clean)=arch/i386/boot
786
787   When "make clean" is executed, make will descend down in arch/i386/boot,
788   and clean as usual. The Makefile located in arch/i386/boot/ may use
789   the subdir- trick to descend further down.
790
791   Note 1: arch/$(ARCH)/Makefile cannot use "subdir-", because that file is
792   included in the top level makefile, and the kbuild infrastructure
793   is not operational at that point.
794
795   Note 2: All directories listed in core-y, libs-y, drivers-y and net-y will
796   be visited during "make clean".
797
798   === 6 Architecture Makefiles
799
800   The top level Makefile sets up the environment and does the preparation,
801   before starting to descend down in the individual directories.
802   The top level makefile contains the generic part, whereas
803   arch/$(ARCH)/Makefile contains what is required to set up kbuild
804   for said architecture.
805   To do so, arch/$(ARCH)/Makefile sets up a number of variables and defines
806   a few targets.
807
808   When kbuild executes, the following steps are followed (roughly):
809   1) Configuration of the kernel => produce .config
```

```
2) Store kernel version in include/linux/version.h
3) Symlink include/asm to include/asm-$(ARCH)
4) Updating all other prerequisites to the target prepare:
   - Additional prerequisites are specified in arch/$(ARCH)/Makefile
5) 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/$(ARCH)/Makefile.
6) All object files are then linked and the resulting file vmlinux is
   located at the root of the obj tree.
   The very first objects linked are listed in head-y, assigned by
   arch/$(ARCH)/Makefile.
7) 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


--- 6.1 Set variables to tweak the build to the architecture

    LDFLAGS          Generic $(LD) options

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

       Example:
              #arch/s390/Makefile
              LDFLAGS         := -m elf_s390
       Note: ldflags-y can be used to further customise
       the flags used. See chapter 3.7.

    LDFLAGS_MODULE   Options for $(LD) when linking modules

       LDFLAGS_MODULE is used to set specific flags for $(LD) when
       linking the .ko files used for modules.
       Default is "-r", for relocatable output.

    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/i386/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.
```

```
863          Example:
864                #arch/s390/Makefile
865                OBJCOPYFLAGS := -O binary
866
867                #arch/s390/boot/Makefile
868                $(obj)/image: vmlinux FORCE
869                       $(call if_changed,objcopy)
870
871          In this example, the binary $(obj)/image is a binary version of
872          vmlinux. The usage of $(call if_changed,xxx) will be described later.
873
874       KBUILD_AFLAGS           $(AS) assembler flags
875
876          Default value - see top level Makefile
877          Append or modify as required per architecture.
878
879          Example:
880                #arch/sparc64/Makefile
881                KBUILD_AFLAGS += -m64 -mcpu=ultrasparc
882
883       KBUILD_CFLAGS           $(CC) compiler flags
884
885          Default value - see top level Makefile
886          Append or modify as required per architecture.
887
888          Often, the KBUILD_CFLAGS variable depends on the configuration.
889
890          Example:
891                #arch/i386/Makefile
892                cflags-$(CONFIG_M386) += -march=i386
893                KBUILD_CFLAGS += $(cflags-y)
894
895          Many arch Makefiles dynamically run the target C compiler to
896          probe supported options:
897
898                #arch/i386/Makefile
899
900                ...
901                cflags-$(CONFIG_MPENTIUMII)     += $(call cc-option,\
902                                        -march=pentium2,-march=i686)
903                ...
904                # Disable unit-at-a-time mode ...
905                KBUILD_CFLAGS += $(call cc-option,-fno-unit-at-a-time)
906                ...
907
908
909          The first example utilises the trick that a config option expands
910          to 'y' when selected.
911
912       CFLAGS_KERNEL    $(CC) options specific for built-in
913
914          $(CFLAGS_KERNEL) contains extra C compiler flags used to compile
915          resident kernel code.
```

CFLAGS_MODULE $(CC) options specific for modules

$(CFLAGS_MODULE) contains extra C compiler flags used to compile code
for loadable kernel modules.

--- 6.2 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 describe how kbuild supports
generating offset header files.

--- 6.3 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.

head-y, init-y, core-y, libs-y, drivers-y, net-y

$(head-y) lists objects to be linked first in vmlinux.
$(libs-y) lists directories where a lib.a archive can be located.
The rest list directories where a built-in.o object file can be
located.

$(init-y) objects will be located after $(head-y).
Then the rest follows in this order:
$(core-y), $(libs-y), $(drivers-y) and $(net-y).

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

Example:

```
#arch/sparc64/Makefile
core-y += arch/sparc64/kernel/
libs-y += arch/sparc64/prom/ arch/sparc64/lib/
drivers-$(CONFIG_OPROFILE)  += arch/sparc64/oprofile/
```

--- 6.4 Architecture-specific boot images

```
969
970           An arch Makefile specifies goals that take the vmlinux file, compress
971           it, wrap it in bootstrapping code, and copy the resulting files
972           somewhere. This includes various kinds of installation commands.
973           The actual goals are not standardized across architectures.
974
975           It is common to locate any additional processing in a boot/
976           directory below arch/$(ARCH)/.
977
978           Kbuild does not provide any smart way to support building a
979           target specified in boot/. Therefore arch/$(ARCH)/Makefile shall
980           call make manually to build a target in boot/.
981
982           The recommended approach is to include shortcuts in
983           arch/$(ARCH)/Makefile, and use the full path when calling down
984           into the arch/$(ARCH)/boot/Makefile.
985
986           Example:
987                   #arch/i386/Makefile
988                   boot := arch/i386/boot
989                   bzImage: vmlinux
990                           $(Q)$(MAKE) $(build)=$(boot) $(boot)/$@
991
992           "$(Q)$(MAKE) $(build)=<dir>" is the recommended way to invoke
993           make in a subdirectory.
994
995           There are no rules for naming architecture-specific targets,
996           but executing "make help" will list all relevant targets.
997           To support this, $(archhelp) must be defined.
998
999           Example:
1000                  #arch/i386/Makefile
1001                  define archhelp
1002                    echo  '* bzImage      - Image (arch/$(ARCH)/boot/bzImage)'
1003                  endif
1004
1005          When make is executed without arguments, the first goal encountered
1006          will be built. In the top level Makefile the first goal present
1007          is all:.
1008          An architecture shall always, per default, build a bootable image.
1009          In "make help", the default goal is highlighted with a '*'.
1010          Add a new prerequisite to all: to select a default goal different
1011          from vmlinux.
1012
1013          Example:
1014                  #arch/i386/Makefile
1015                  all: bzImage
1016
1017          When "make" is executed without arguments, bzImage will be built.
1018
1019  --- 6.5 Building non-kbuild targets
1020
1021      extra-y
```

```
1022
1023          extra-y specify additional targets created in the current
1024          directory, in addition to any targets specified by obj-*.
1025
1026          Listing all targets in extra-y is required for two purposes:
1027          1) Enable kbuild to check changes in command lines
1028             - When $(call if_changed,xxx) is used
1029          2) kbuild knows what files to delete during "make clean"
1030
1031          Example:
1032                 #arch/i386/kernel/Makefile
1033                 extra-y := head.o init_task.o
1034
1035          In this example, extra-y is used to list object files that
1036          shall be built, but shall not be linked as part of built-in.o.
1037
1038
1039   --- 6.6 Commands useful for building a boot image
1040
1041          Kbuild provides a few macros that are useful when building a
1042          boot image.
1043
1044       if_changed
1045
1046          if_changed is the infrastructure used for the following commands.
1047
1048          Usage:
1049                 target: source(s) FORCE
1050                        $(call if_changed,ld/objcopy/gzip)
1051
1052          When the rule is evaluated, it is checked to see if any files
1053          need an update, or the command line has changed since the last
1054          invocation. The latter will force a rebuild if any options
1055          to the executable have changed.
1056          Any target that utilises if_changed must be listed in $(targets),
1057          otherwise the command line check will fail, and the target will
1058          always be built.
1059          Assignments to $(targets) are without $(obj)/ prefix.
1060          if_changed may be used in conjunction with custom commands as
1061          defined in 6.7 "Custom kbuild commands".
1062
1063          Note: It is a typical mistake to forget the FORCE prerequisite.
1064          Another common pitfall is that whitespace is sometimes
1065          significant; for instance, the below will fail (note the extra space
1066          after the comma):
1067                 target: source(s) FORCE
1068          #WRONG!#      $(call if_changed, ld/objcopy/gzip)
1069
1070       ld
1071          Link target. Often, LDFLAGS_$@ is used to set specific options to ld.
1072
1073       objcopy
1074          Copy binary. Uses OBJCOPYFLAGS usually specified in
```

```
1075          arch/$(ARCH)/Makefile.
1076          OBJCOPYFLAGS_$@ may be used to set additional options.
1077
1078       gzip
1079          Compress target. Use maximum compression to compress target.
1080
1081          Example:
1082                 #arch/i386/boot/Makefile
1083                 LDFLAGS_bootsect := -Ttext 0x0 -s --oformat binary
1084                 LDFLAGS_setup    := -Ttext 0x0 -s --oformat binary -e begtext
1085
1086                 targets += setup setup.o bootsect bootsect.o
1087                 $(obj)/setup $(obj)/bootsect: %: %.o FORCE
1088                        $(call if_changed,ld)
1089
1090          In this example, there are two possible targets, requiring different
1091          options to the linker. The linker options are specified using the
1092          LDFLAGS_$@ syntax - one for each potential target.
1093          $(targets) are assigned all potential targets, by which kbuild knows
1094          the targets and will:
1095                 1) check for commandline changes
1096                 2) delete target during make clean
1097
1098          The ": %: %.o" part of the prerequisite is a shorthand that
1099          free us from listing the setup.o and bootsect.o files.
1100          Note: It is a common mistake to forget the "target :=" assignment,
1101                resulting in the target file being recompiled for no
1102                obvious reason.
1103
1104
1105    --- 6.7 Custom kbuild commands
1106
1107          When kbuild is executing with KBUILD_VERBOSE=0, then only a shorthand
1108          of a command is normally displayed.
1109          To enable this behaviour for custom commands kbuild requires
1110          two variables to be set:
1111          quiet_cmd_<command>  - what shall be echoed
1112                cmd_<command>  - the command to execute
1113
1114          Example:
1115                 #
1116                 quiet_cmd_image = BUILD   $@
1117                       cmd_image = $(obj)/tools/build $(BUILDFLAGS) \
1118                                                   $(obj)/vmlinux.bin > $@
1119
1120                 targets += bzImage
1121                 $(obj)/bzImage: $(obj)/vmlinux.bin $(obj)/tools/build FORCE
1122                        $(call if_changed,image)
1123                        @echo 'Kernel: $@ is ready'
1124
1125          When updating the $(obj)/bzImage target, the line
1126
1127          BUILD    arch/i386/boot/bzImage
```

```
1128
1129          will be displayed with "make KBUILD_VERBOSE=0".
1130
1131
1132    --- 6.8 Preprocessing linker scripts
1133
1134          When the vmlinux image is built, the linker script
1135          arch/$(ARCH)/kernel/vmlinux.lds is used.
1136          The script is a preprocessed variant of the file vmlinux.lds.S
1137          located in the same directory.
1138          kbuild knows .lds files and includes a rule *lds.S -> *lds.
1139
1140          Example:
1141                #arch/i386/kernel/Makefile
1142                always := vmlinux.lds
1143
1144                #Makefile
1145                export CPPFLAGS_vmlinux.lds += -P -C -U$(ARCH)
1146
1147          The assignment to $(always) is used to tell kbuild to build the
1148          target vmlinux.lds.
1149          The assignment to $(CPPFLAGS_vmlinux.lds) tells kbuild to use the
1150          specified options when building the target vmlinux.lds.
1151
1152          When building the *.lds target, kbuild uses the variables:
1153          KBUILD_CPPFLAGS      : Set in top-level Makefile
1154          cppflags-y    : May be set in the kbuild makefile
1155          CPPFLAGS_$(@F)  : Target specific flags.
1156                            Note that the full filename is used in this
1157                            assignment.
1158
1159          The kbuild infrastructure for *lds file are used in several
1160          architecture-specific files.
1161
1162    === 7 Kbuild syntax for exported headers
1163
1164    The kernel include a set of headers that is exported to userspace.
1165    Many headers can be exported as-is but other headers requires  a
1166    minimal pre-processing before they are ready for user-space.
1167    The pre-processing does:
1168    - drop kernel specific annotations
1169    - drop include of compiler.h
1170    - drop all sections that is kernel internat (guarded by ifdef __KERNEL__)
1171
1172    Each relevant directory contain a file name "Kbuild" which specify the
1173    headers to be exported.
1174    See subsequent chapter for the syntax of the Kbuild file.
1175
1176          --- 7.1 header-y
1177
1178          header-y specify header files to be exported.
1179
1180                Example:
```

```
                        #include/linux/Kbuild
                        header-y += usb/
                        header-y += aio_abi.h

        The convention is to list one file per line and
        preferably in alphabetic order.

        header-y also specify which subdirectories to visit.
        A subdirectory is identified by a trailing '/' which
        can be seen in the example above for the usb subdirectory.

        Subdirectories are visited before their parent directories.

        --- 7.2 objhdr-y

        objhdr-y specifies generated files to be exported.
        Generated files are special as they need to be looked
        up in another directory when doing 'make O=...' builds.

               Example:
                        #include/linux/Kbuild
                        objhdr-y += version.h

        --- 7.3 destination-y

        When an architecture have a set of exported headers that needs to be
        exported to a different directory destination-y is used.
        destination-y specify the destination directory for all exported
        headers in the file where it is present.

               Example:
                        #arch/xtensa/platforms/s6105/include/platform/Kbuild
                        destination-y := include/linux

        In the example above all exported headers in the Kbuild file
        will be located in the directory "include/linux" when exported.


        --- 7.4 unifdef-y (deprecated)

        unifdef-y is deprecated. A direct replacement is header-y.


=== 8 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 ...
```

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, 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 will used as the option(s) to the strip command.

=== 9 Makefile language

1287
1288 The kernel Makefiles are designed to be run with GNU Make. The Makefiles
1289 use only the documented features of GNU Make, but they do use many
1290 GNU extensions.
1291
1292 GNU Make supports elementary list-processing functions. The kernel
1293 Makefiles use a novel style of list building and manipulation with few
1294 "if" statements.
1295
1296 GNU Make has two assignment operators, ":=" and "=". ":=" performs
1297 immediate evaluation of the right-hand side and stores an actual string
1298 into the left-hand side. "=" is like a formula definition; it stores the
1299 right-hand side in an unevaluated form and then evaluates this form each
1300 time the left-hand side is used.
1301
1302 There are some cases where "=" is appropriate. Usually, though, ":="
1303 is the right choice.
1304
1305 === 10 Credits
1306
1307 Original version made by Michael Elizabeth Chastain, <mailto:mec@shout.net>
1308 Updates by Kai Germaschewski <kai@tp1.ruhr-uni-bochum.de>
1309 Updates by Sam Ravnborg <sam@ravnborg.org>
1310 Language QA by Jan Engelhardt <jengelh@gmx.de>
1311
1312 === 11 TODO
1313
1314 - Describe how kbuild supports shipped files with _shipped.
1315 - Generating offset header files.
1316 - Add more variables to section 7?
1317
1318
1319

【附录 F】modules.txt

1
2 In this document you will find information about:
3 - how to build external modules
4 - how to make your module use the kbuild infrastructure
5 - how kbuild will install a kernel
6 - how to install modules in a non-standard location
7
8 === Table of Contents
9
10

=== 1. Introduction

kbuild includes functionality for building modules both
within the kernel source tree and outside the kernel source tree.
The latter is usually referred to as external or "out-of-tree"
modules and is used both during development and for modules that
are not planned to be included in the kernel tree.

What is covered within this file is mainly information to authors
of 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. A complete example will be presented in
chapter 4, "Creating a kbuild file for an external module".

=== 2. How to build external modules

kbuild offers functionality to build external modules, with the
prerequisite that there is a pre-built kernel available with full source.
A subset of the targets available when building the kernel is available
when building an external module.

--- 2.1 Building external modules

Use the following command to build an external module:

```
make -C <path-to-kernel> M=`pwd`
```

For the running kernel use:

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

For the above command to succeed, the kernel must have been
built with modules enabled.

To install the modules that were just built:

```
72                make -C <path-to-kernel> M=`pwd` modules_install
73
74          More complex examples will be shown later, the above should
75          be enough to get you started.
76
77    --- 2.2 Available targets
78
79          $KDIR refers to the path to the kernel source top-level directory
80
81          make -C $KDIR M=`pwd`
82                 Will build the module(s) located in current directory.
83                 All output files will be located in the same directory
84                 as the module source.
85                 No attempts are made to update the kernel source, and it is
86                 a precondition that a successful make has been executed
87                 for the kernel.
88
89          make -C $KDIR M=`pwd` modules
90                 The modules target is implied when no target is given.
91                 Same functionality as if no target was specified.
92                 See description above.
93
94          make -C $KDIR M=`pwd` modules_install
95                 Install the external module(s).
96                 Installation default is in /lib/modules/<kernel-version>/extra,
97                 but may be prefixed with INSTALL_MOD_PATH - see separate
98                 chapter.
99
100         make -C $KDIR M=`pwd` clean
101                Remove all generated files for the module - the kernel
102                source directory is not modified.
103
104         make -C $KDIR M=`pwd` help
105                help will list the available target when building external
106                modules.
107
108   --- 2.3 Available options:
109
110         $KDIR refers to the path to the kernel source top-level directory
111
112         make -C $KDIR
113                Used to specify where to find the kernel source.
114                '$KDIR' represent the directory where the kernel source is.
115                Make will actually change directory to the specified directory
116                when executed but change back when finished.
117
118         make -C $KDIR M=`pwd`
119                M= is used to tell kbuild that an external module is
120                being built.
121                The option given to M= is the directory where the external
122                module (kbuild file) is located.
123                When an external module is being built only a subset of the
124                usual targets are available.
```

```
125
126         make -C $KDIR SUBDIRS=`pwd`
127                Same as M=. The SUBDIRS= syntax is kept for backwards
128                compatibility.
129
130    --- 2.4 Preparing the kernel tree for module build
131
132         To make sure the kernel contains the information required to
133         build external modules the target 'modules_prepare' must be used.
134         'modules_prepare' exists solely as a simple way to prepare
135         a kernel source tree for building external modules.
136         Note: modules_prepare will not build Module.symvers even if
137         CONFIG_MODVERSIONS is set. Therefore a full kernel build
138         needs to be executed to make module versioning work.
139
140    --- 2.5 Building separate files for a module
141         It is possible to build single files which are part of a module.
142         This works equally well for the kernel, a module and even for
143         external modules.
144         Examples (module foo.ko, consist of bar.o, baz.o):
145                make -C $KDIR M=`pwd` bar.lst
146                make -C $KDIR M=`pwd` bar.o
147                make -C $KDIR M=`pwd` foo.ko
148                make -C $KDIR M=`pwd` /
149
150
151    === 3. Example commands
152
153    This example shows the actual commands to be executed when building
154    an external module for the currently running kernel.
155    In the example below, the distribution is supposed to use the
156    facility to locate output files for a kernel compile in a different
157    directory than the kernel source - but the examples will also work
158    when the source and the output files are mixed in the same directory.
159
160    # Kernel source
161    /lib/modules/<kernel-version>/source -> /usr/src/linux-<version>
162
163    # Output from kernel compile
164    /lib/modules/<kernel-version>/build -> /usr/src/linux-<version>-up
165
166    Change to the directory where the kbuild file is located and execute
167    the following commands to build the module:
168
169         cd /home/user/src/module
170         make -C /usr/src/`uname -r`/source            \
171                 O=/lib/modules/`uname-r`/build        \
172                 M=`pwd`
173
174    Then, to install the module use the following command:
175
176         make -C /usr/src/`uname -r`/source            \
177                 O=/lib/modules/`uname-r`/build        \
```

```
                 M=`pwd`                                \
                modules_install

If you look closely you will see that this is the same command as
listed before - with the directories spelled out.

The above are rather long commands, and the following chapter
lists a few tricks to make it all easier.


=== 4. Creating a kbuild file for an external module

kbuild is the build system for the kernel, and external modules
must use kbuild to stay compatible with changes in the build system
and to pick up the right flags to gcc etc.

The kbuild file used as input shall follow the syntax described
in Documentation/kbuild/makefiles.txt. This chapter will introduce a few
more tricks to be used when dealing with external modules.

In the following a Makefile will be created for a module with the
following files:
       8123_if.c
       8123_if.h
       8123_pci.c
       8123_bin.o_shipped    <= Binary blob

--- 4.1 Shared Makefile for module and kernel

       An external module always includes a wrapper Makefile supporting
       building the module using 'make' with no arguments.
       The Makefile provided will most likely include additional
       functionality such as test targets etc. and this part shall
       be filtered away from kbuild since it may impact kbuild if
       name clashes occurs.

       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

              KERNELDIR := /lib/modules/`uname -r`/build
              all::
                     $(MAKE) -C $(KERNELDIR) M=`pwd` $@

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

```
231
232             endif
233
234         In example 1, the check for KERNELRELEASE is used to separate
235         the two parts of the Makefile. kbuild will only see the two
236         assignments whereas make will see everything except the two
237         kbuild assignments.
238
239         In recent versions of the kernel, kbuild will look for a file named
240         Kbuild and as second option look for a file named Makefile.
241         Utilising the Kbuild file makes us split up the Makefile in example 1
242         into two files as shown in example 2:
243
244         Example 2:
245             --> filename: Kbuild
246             obj-m  := 8123.o
247             8123-y := 8123_if.o 8123_pci.o 8123_bin.o
248
249             --> filename: Makefile
250             KERNELDIR := /lib/modules/`uname -r`/build
251             all::
252                     $(MAKE) -C $(KERNELDIR) M=`pwd` $@
253
254             # Module specific targets
255             genbin:
256                     echo "X" > 8123_bin.o_shipped
257
258
259         In example 2, we are down to two fairly simple files and for simple
260         files as used in this example the split is questionable. But some
261         external modules use Makefiles of several hundred lines and here it
262         really pays off to separate the kbuild part from the rest.
263         Example 3 shows a backward compatible version.
264
265         Example 3:
266             --> filename: Kbuild
267             obj-m  := 8123.o
268             8123-y := 8123_if.o 8123_pci.o 8123_bin.o
269
270             --> filename: Makefile
271             ifneq ($(KERNELRELEASE),)
272             include Kbuild
273             else
274             # Normal Makefile
275
276             KERNELDIR := /lib/modules/`uname -r`/build
277             all::
278                     $(MAKE) -C $(KERNELDIR) M=`pwd` $@
279
280             # Module specific targets
281             genbin:
282                     echo "X" > 8123_bin.o_shipped
283
```

```
284                endif
285
286          The trick here is to include the Kbuild file from Makefile, so
287          if an older version of kbuild picks up the Makefile, the Kbuild
288          file will be included.
289
290    --- 4.2 Binary blobs included in a module
291
292          Some external modules needs to include a .o as a blob. kbuild
293          has support for this, but requires the blob file to be named
294          <filename>_shipped. In our example the blob is named
295          8123_bin.o_shipped and when the kbuild rules kick in the file
296          8123_bin.o is created as a simple copy off the 8213_bin.o_shipped file
297          with the _shipped part stripped of the filename.
298          This allows the 8123_bin.o filename to be used in the assignment to
299          the module.
300
301          Example 4:
302                obj-m  := 8123.o
303                8123-y := 8123_if.o 8123_pci.o 8123_bin.o
304
305          In example 4, there is no distinction between the ordinary .c/.h files
306          and the binary file. But kbuild will pick up different rules to create
307          the .o file.
308
309
310    === 5. Include files
311
312    Include files are a necessity when a .c file uses something from other .c
313    files (not strictly in the sense of C, but if good programming practice is
314    used). Any module that consists of more than one .c file will have a .h file
315    for one of the .c files.
316
317    - If the .h file only describes a module internal interface, then the .h file
318      shall be placed in the same directory as the .c files.
319    - If the .h files describe an interface used by other parts of the kernel
320      located in different directories, the .h files shall be located in
321      include/linux/ or other include/ directories as appropriate.
322
323    One exception for this rule is larger subsystems that have their own directory
324    under include/ such as include/scsi. Another exception is arch-specific
325    .h files which are located under include/asm-$(ARCH)/*.
326
327    External modules have a tendency to locate include files in a separate include/
328    directory and therefore need to deal with this in their kbuild file.
329
330    --- 5.1 How to include files from the kernel include dir
331
332          When a module needs to include a file from include/linux/, then one
333          just uses:
334
335                #include <linux/modules.h>
336
```

```
337          kbuild will make sure to add options to gcc so the relevant
338          directories are searched.
339          Likewise for .h files placed in the same directory as the .c file.
340
341                #include "8123_if.h"
342
343          will do the job.
344
345    --- 5.2 External modules using an include/ dir
346
347          External modules often locate their .h files in a separate include/
348          directory although this is not usual kernel style. When an external
349          module uses an include/ dir then kbuild needs to be told so.
350          The trick here is to use either EXTRA_CFLAGS (take effect for all .c
351          files) or CFLAGS_$F.o (take effect only for a single file).
352
353          In our example, if we move 8123_if.h to a subdirectory named include/
354          the resulting Kbuild file would look like:
355
356                --> filename: Kbuild
357                obj-m  := 8123.o
358
359                EXTRA_CFLAGS := -Iinclude
360                8123-y := 8123_if.o 8123_pci.o 8123_bin.o
361
362          Note that in the assignment there is no space between -I and the path.
363          This is a kbuild limitation:  there must be no space present.
364
365    --- 5.3 External modules using several directories
366
367          If an external module does not follow the usual kernel style, but
368          decides to spread files over several directories, then kbuild can
369          handle this too.
370
371          Consider the following example:
372
373          |
374          +- src/complex_main.c
375          |   +- hal/hardwareif.c
376          |   +- hal/include/hardwareif.h
377          +- include/complex.h
378
379          To build a single module named complex.ko, we then need the following
380          kbuild file:
381
382          Kbuild:
383                obj-m := complex.o
384                complex-y := src/complex_main.o
385                complex-y += src/hal/hardwareif.o
386
387                EXTRA_CFLAGS := -I$(src)/include
388                EXTRA_CFLAGS += -I$(src)src/hal/include
389
```

kbuild knows how to handle .o files located in another directory -
although this is NOT recommended practice. The syntax is to specify
the directory relative to the directory where the Kbuild file is
located.

To find the .h files, we have to explicitly tell kbuild where to look
for the .h files. When kbuild executes, the current directory is always
the root of the kernel tree (argument to -C) and therefore we have to
tell kbuild how to find the .h files using absolute paths.
$(src) will specify the absolute path to the directory where the
Kbuild file are located when being build as an external module.
Therefore -I$(src)/ is used to point out the directory of the Kbuild
file and any additional path are just appended.

=== 6. Module installation

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

	/lib/modules/$(KERNELRELEASE)/kernel

External modules are installed in the directory:

	/lib/modules/$(KERNELRELEASE)/extra

--- 6.1 INSTALL_MOD_PATH

Above are the default directories, but as always, some level of
customization is possible. One can prefix the 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 in the
example above, can be specified on the command line when calling make.
INSTALL_MOD_PATH has effect both when installing modules included in
the kernel as well as when installing external modules.

--- 6.2 INSTALL_MOD_DIR

When installing external modules they are by default installed to a
directory under /lib/modules/$(KERNELRELEASE)/extra, but one may wish
to locate modules for a specific functionality in a separate
directory. For this purpose, one can use INSTALL_MOD_DIR to specify an
alternative name to 'extra'.

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

=== 7. Module versioning & Module.symvers

Module versioning is enabled by the CONFIG_MODVERSIONS tag.

Module versioning is used as a simple ABI consistency check. The Module versioning creates a CRC value of the full prototype for an exported symbol and when a module is loaded/used then the CRC values contained in the kernel are compared with similar values in the module. If they are not equal, then the kernel refuses to load the module.

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

--- 7.1 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 symbols, the corresponding CRC value is stored too.

The syntax of the Module.symvers file is:

<CRC> <Symbol> <module>

Sample:

0x2d036834 scsi_remove_host drivers/scsi/scsi_mod

For a kernel build without CONFIG_MODVERSIONS enabled, the crc would read: 0x00000000

Module.symvers serves two purposes:
1) It lists all exported symbols both from vmlinux and all modules
2) It lists the CRC if CONFIG_MODVERSIONS is enabled

--- 7.2 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 and to obtain all symbols, modpost reads Module.symvers from the kernel.
If a Module.symvers file is present in the directory where the external module is being built, this file will be read too.
During the MODPOST step, a new Module.symvers file will be written containing all exported symbols that were not defined in the kernel.

--- 7.3 Symbols from another external module

Sometimes, an external module uses exported symbols from another external module. Kbuild needs to have full knowledge on all symbols to avoid spitting out warnings about undefined symbols.
Three solutions exist to let kbuild know all symbols of more than one external module.
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' and 'bar', and 'foo' needs

```
496                symbols from 'bar', then one can use a common top-level kbuild
497                file so both modules are compiled in same build.
498
499                Consider following directory layout:
500                ./foo/ <= contains the foo module
501                ./bar/ <= contains the bar module
502                The top-level Kbuild file would then look like:
503
504                #./Kbuild: (this file may also be named Makefile)
505                       obj-y := foo/ bar/
506
507                Executing:
508                       make -C $KDIR M=`pwd`
509
510                will then do the expected and compile both modules with full
511                knowledge on symbols from both modules.
512
513          Use an extra Module.symvers file
514                When an external module is built, a Module.symvers file is
515                generated containing all exported symbols which are not
516                defined in the kernel.
517                To get access to symbols from module 'bar', one can copy the
518                Module.symvers file from the compilation of the 'bar' module
519                to the directory where the 'foo' module is built.
520                During the module build, kbuild will read the Module.symvers
521                file in the directory of the external module and when the
522                build is finished, a new Module.symvers file is created
523                containing the sum of all symbols defined and not part of the
524                kernel.
525
526          Use make variable KBUILD_EXTRA_SYMBOLS in the Makefile
527                If it is impractical to copy Module.symvers from another
528                module, you can assign a space separated list of files to
529                KBUILD_EXTRA_SYMBOLS in your Makfile. These files will be
530                loaded by modpost during the initialisation of its symbol
531                tables.
532
533    === 8. Tips & Tricks
534
535    --- 8.1 Testing for CONFIG_FOO_BAR
536
537          Modules often need to check for certain CONFIG_ options to decide if
538          a specific feature shall be included in the module. When kbuild is used
539          this is done by referencing the CONFIG_ variable directly.
540
541                #fs/ext2/Makefile
542                obj-$(CONFIG_EXT2_FS) += ext2.o
543
544                ext2-y := balloc.o bitmap.o dir.o
545                ext2-$(CONFIG_EXT2_FS_XATTR) += xattr.o
546
547          External modules have traditionally used grep to check for specific
548          CONFIG_ settings directly in .config. This usage is broken.
```

549 As introduced before, external modules shall use kbuild when building
550 and therefore can use the same methods as in-kernel modules when
551 testing for CONFIG_ definitions.
552