第一部分：

# 1.入门

Buildroot是一款使用交叉编译自动构建完整的Linux嵌入式系统的工具，简化了开发者的工作。

为次，Buildroot能够为您的目标板生成一个交叉编译工具链，一个根文件系统，一个Linux内核映像和一个引导加载程序。 Buildroot可以使用这些选项的组合（例如，可以使用已有的交叉编译工具链，只使用Buildroot构建根文件系统）。

Buildroot主要用于使用嵌入式系统的人员。嵌入式系统通常使用的处理器与通用的x86处理器是不同的。可能是PowerPC处理器，MIPS处理器，ARM处理器等。

Buildroot支持众多处理器及其变体;它还提供了用于现有可用的几个开发板的默认配置。除此之外，许多第3方项目都是基于Buildroot之上的开发BSP或SDK。

# 2． 系统要求

Buildroot运行于linux系统之上。

尽管，Buildroot本身会构建编译所需的大部分主机安装包，但是某些标准的linux工具还是期望提前安装在host机上。下面，你会发现必须的和可选的安装包的概述（请注意，安装包的名称可能会因为发布版本而发生变化）。

## 2.1 必要安装包

### 2.1.1构建工具:

（1）which

（2）sed

（3）make (version 3.81 or any later)

（4）binutils

（5）build-essential (only for Debian based systems)

（6）gcc (version 2.95 or any later)

（7）g++ (version 2.95 or any later)

（8）bash

（9）patch

（10）gzip

（11）bzip2

（12）perl (version 5.8.7 or any later)

（13）tar

（14）cpio

（15）python (version 2.6 or any later)

（16）unzip

（17）rsync

（18）file (must be in /usr/bin/file)

（19）bc

### 2.1.2 下载工具

wget

## 2.2 可选包

### 2.2.1 配置界面依赖工具：

For these libraries, you need to install both runtime and development data, which in many distributions are packaged separately. The development packages typically have a -dev or -devel suffix.

（1）ncurses5使用menuconfig进行配置时

（2）qt4使用xconfig进行配置时

（3）glib2，gtk2 和glade2 使用gconfig配置界面时

### 2.2.2 源代码获取工具

In the official tree, most of the package sources are retrieved using wget from ftp, http or https locations. A few packages are only available through a version control system. Moreover, Buildroot is capable of downloading sources via other tools, like rsync or scp (refer to Chapter 19, Download infrastructure for more details). If you enable packages using any of these methods, you will need to install the corresponding tool on the host system:

（1）bazaar

（2）cvs

（3）git

（4）mercurial

（5）rsync

（6）scp

（7）subversion

### 2.2.3 Java依赖包，如果需要为目标系统构建Java Classpath

（1）javac编译器

（2）jar工具

### 2.2.4 文档生成工具

（1）asciidoc, version 8.6.3 or higher

（2）w3m

（3）python with the argparse module (automatically present in 2.7+ and 3.2+)

（4）dblatex (required for the pdf manual only)

### 2.2.5 图形生成工具

（1）graphviz to use graph-depends and <pkg>-graph-depends

（2）python-matplotlib to use graph-build

# 3. 获取Buildroot

Buildroot版本每3个月发布一次，分别在2月，5月，8月和11月发布。 版本号格式为YYYY.MM，例如2013.02，2014.08。

下载地址：<http://buildroot.org/downloads/>.

# 4. 快速启动Buildroot

重要：普通用户权限即可编译所有的安装包，不需要root权限进行configure或者使用Buildroot。以普通用户权限运行，保护你的host主机不被编译和安装破坏。

进入Buildroot根目录，运行

$ make menuconfig

或者基于原始的命令行配置，

$ make nconfig

基于新的命令行配置，

$ make xconfig

基于qt配置，

$ make gconfig

一旦配置完成，配置工具将生成一个.config文件。 该文件将由顶层Makefile读取。

启动编译：

$ make

对于Buildroot来说，你**绝对不能**使用make -jN:目前顶层并行make并不支持。代替的，可以使用BR2\_JLEVE选项告诉Buildroot使用make -jN编译每个独立的包。

通常，make会执行下面的步骤：

1. 下载源文件（需要的话）；
2. configure，build和install交叉编译工具，或者简单地导入外部工具链；
3. configure，build和install选择的目标安装包；
4. 构建kernel镜像，如果选择的话；
5. 构建bootloader镜像，如果选择的话；
6. 创建root文件系统，用选择的格式。

Buildroot输出都会被放到output目录下，这个目录包含下面几个子目录：

1. images/ 存放（kernel, bootloader和rootfs）。这些文件都是要放置到你的目标板上的。
2. build/ 该目录下所有的组件都会被build（包括Buildroot需要的关于主机和为目标板编译的安装包）。每一个组件一个目录。
3. staging 类似于根文件系统的结构。这个目录包含交叉编译工具的头文件和库文件，以及为目标板选择的所有用户空间的安装包。但是，这个目录不是打算成为目标板的root文件系统：它包含了许多开发文件，unstripped 二进制文件和库，对于嵌入式系统来说，太庞大了。这些开发文件被用来为依赖其它库的目标板编译库文件和应用程序。
4. target 包含几乎完整的目标板rootfs：除了/dev目录下的内容以外都有。（Buildroot不能产生它们，因为Buildroot不能也不想运行root权限）。另外，也不会产生许可权限（例如，setuid为busybox 二进制文件）。因此，这个目录不应该在你的目标板上使用。相较staging，target目录而言，开发文件没有，二进制文件也是被striped过的。
5. host目录，为了Buildroot正确执行而在主机上安装的一个工具，包括交叉工具链。

# 5. 社区资源

（1）邮件列表；

（2）IRC

（3）Bug tracker

（4）Wiki

（5）Patchwork

第二部分 用户指导

# 6. 配置Buildroot

命令make \*config产生的所有配置选项，都会提供详细的帮助信息。

命令make \*config也提供了一个搜索工具。

在menuconfig中，搜索工具通过按下/进行调用；

在xconfig中，搜索工具通过按下ctrl+f进行调用。

搜索的结果就是对应项的帮助信息。在menuconfig配置界面里。在menuconfig中，左列中的数字提供了相应条目的快捷方式。只需输入该数字即可直接跳转到条目，或者在条目由于缺失依赖项而无法选择的情况下跳转到其包含菜单。

虽然条目的菜单结构和帮助文本应该充分说明，但有些主题需要额外的解释，但不能在帮助文本中轻松涵盖，因此将在以下各节中进行介绍。

## 6.1 交叉编译工具链

编译工具链是为你的系统编译代码而提供的工具集。包含编译器（例如，gcc）像汇编器和链接器（例，binutils）和C标准库（例如，GNU libc，uClibc-ng）。

安装在您的开发工作站上的系统当然已经有一个编译工具链，您可以使用它编译在您的系统上运行的应用程序。如果您使用PC，则编译工具链将在x86处理器上运行，并为x86处理器生成代码。在大多数Linux系统下，编译工具链使用GNU libc（glibc）作为C标准库。这个编译工具链被称为“主机编译工具链”。运行在机器上，你正在工作的系统，称为“主机系统”[3]。

这个编译工具链由你的发行版提供，Buildroot与它无关（除了使用它来构建在开发主机上运行的交叉编译工具链和其他工具）

如上所述，系统附带的编译工具链运行并为主机系统中的处理器生成代码。由于您的嵌入式系统具有不同的处理器，您需要一个交叉编译工具链 - 一个编译工具链，该编译工具链在您的主机系统上运行，但会为您的目标系统（和目标处理器）生成代码。例如，如果您的主机系统使用x86，而您的目标系统使用ARM，则主机上的常规编译工具链将在x86上运行，并为x86生成代码，而交叉编译工具链将在x86上运行并生成ARM代码。

Buildroot为交叉编译工具链提供了两种解决方案：

内部工具链，在配置界面中称为Buildroot工具链。

外部工具链，在配置界面中称为外部工具链。

这两个解决方案之间的选择是使用Toolchain菜单中的Toolchain Type选项完成的。一旦选择了一种解决方案，就会出现一些配置选项，在下面的章节中详细介绍

### 6.1.1 内部工具链

### 6.1.2 外部工具链

### 6.1.3 外部工具链封装程序

## 6.2 /dev目录的管理

在Linux系统上，/dev目录包含特殊文件，称为设备文件，允许用户空间应用程序访问由Linux内核管理的硬件设备。如果没有这些设备文件，即使用户空间应用程序能够被Linux内核正确识别，用户空间应用程序也将无法使用硬件设备。

在System Configuation，/dev management中，Buildroot提供了四种不同的解决方案来处理/dev目录：

第一种解决方案就是静态使用设备表。这是linux中以前使用的方法。使用这种方法，设备文件永久存储在根文件系统中（即它们在重新启动时持续存在），并且当硬件设备被添加或从系统中移除时，没有任何东西会自动创建和删除这些设备文件。因此，Buildroot使用设备表创建一组标准设备文件，默认存储在Buildroot源代码的system / device\_table\_dev.txt中。当Buildroot生成最终的根文件系统映像时会处理此文件，因此设备文件在output / target目录中不可见。BR2\_ROOTFS\_STATIC\_DEVICE\_TABLE选项允许更改Buildroot使用的默认设备表，或添加额外的设备表，以便在构建过程中由Buildroot创建其他设备文件。因此，如果您使用此方法，并且系统中缺少设备文件，则可以创建包含其他设备文件说明的board / <公司名称> / <项目名称> /device\_table\_dev.txt文件，然后您可以将BR2\_ROOTFS\_STATIC\_DEVICE\_TABLE设置为system / device\_table\_dev.txt board / <公司> / <项目> /device\_table\_dev.txt。有关设备表格文件格式的更多详细信息，请参见第23章Makedev语法文档。

第二个解决方案是仅动态使用devtmpfs。 devtmpfs是内核2.6.32中引入的Linux内核中的虚拟文件系统（如果您使用的是较旧的内核，则无法使用此选项）。当挂载到/dev中时，此虚拟文件系统将自动使设备文件出现和消除，因为硬件设备将从系统中添加和删除。这个文件系统在重新启动时并不持续：它由内核动态填充。使用devtmpfs需要启用以下内核配置选项：CONFIG\_DEVTMPFS和CONFIG\_DEVTMPFS\_MOUNT。当Buildroot负责为您的嵌入式设备构建Linux内核时，它确保启用这两个选项。但是，如果您在Buildroot之外构建Linux内核，那么必须选择这两个选项（如果未能这样做， Buildroot系统将无法boot）。

第三个解决方案：动态使用devtmpfs+mdev。当有硬件设备接入时，动态添加删除设备文件。编译内核时需要配置CONFIG\_DEVTMPFS和CONFIG\_DEVTMPFS\_MOUNT两个选项。mdev是在其之上的用户空间工具。mdev是Busybox的一部分，kernel每次添加或者移除器件时，都会调用它。/etc/mdev.conf配置文件，可以实现许可权和拥有权限的修改，也可以在设备添加删除时，调用某个脚本。本质上说，它允许用户空间能够与设备的添加移除事件进行交互。比如说，当设备添加到系统中时，mdev会自动载入内核。如果有需要固件的设备，mdev负责吧固件程序推送给kernel。mdev是udev的轻量级实现。更多关于mdev的详细信息和它配置文件的语法可以参考下面的文章：<http://git.busybox.net/busybox/tree/docs/mdev.txt>.

第四种方案是，动态使用devtmpfs+eudev。eudev是建立在devtmpfs之上的用户空间守护进程。eudev是运行在后台的守护进程，当有一个设备添加或者移除时，kernel会调用。这是一个比mdev更为重量级的解决方案，但是提供了更大的便利性。eudev是udev的独立版本，udev是大多数桌面Linux发行版中使用的原始用户空间守护进程，现在它已成为Systemd的一部分。更多详细信息请参考下面的网址：<http://en.wikipedia.org/wiki/Udev>.

Buildroot开发人员的建议是，首先仅动态使用devtmpfs，直到您需要在添加/移除设备时或需要固件时通知用户空间，在这种情况下，使用devtmpfs+mdev的动态通常是更好的解决方案。

请注意，如果选择systemd作为init系统，/dev管理将会通过systemd提供的udev程序实施。

## 6.3 init系统

init程序是内核启动的第一个用户空间程序（其进程号是1），负责启动用户空间服务和程序（例如：web服务器，图形程序，其它网络服务等。）。

Build允许使用不同类型的init系统，可以通过System Configuration，Init system中选择：

第一种解决方案是busybox。在许多程序中，BusyBox有一个基本的init程序的实现，这对大多数嵌入式系统来说都是足够的。启用BR2\_INIT\_BUSYBOX将确保BusyBox将构建并安装其init程序。当然了，这在Buildroot中是默认选项。BusyBox的init程序将在启动时读取/etc/ inittab文件以知道做什么。 可以在

<http://git.busybox.net/busybox/tree/examples/inittab>

中找到该文件的语法（请注意，BusyBox inittab语法很特殊：不要使用Internet上的随机inittab文档来了解BusyBox inittab）。 Buildroot中的默认inittab存储在system/ skeleton/etc/inittab中。 除了安装一些重要的文件系统外，默认的inittab所执行的主要工作是启动/etc/init.d/rcS脚本，并启动一个getty程序（它提供了一个登录提示符）。

第二个解决方案是systemV。这个解决方案使用传统的sysvinit程序，打包在Buildroot中的package/sysvinit里。这是大多数桌面Linux发行版中使用的解决方案，直到他们转向Upstart或Systemd等更新的替代方案。 sysvinit也可以使用inittab文件（与BusyBox的语法略有不同）。 使用此init解决方案安装的默认inittab位于package/sysvinit/inittab中。

第三个解决方案是systemd。其是linux新一代的init系统。它功能比早先的init系统更多：积极的并行化功能，使用套接字和D-Bus激活来启动服务，按需启动守护进程，跟踪使用Linux控制组的程序，支持快照和恢复系统状态等。值得注意的是，systemd也带来了大量的依赖关系：dbus，udev等等。关于systemd的更多详细信息，请参考：

<http://www.freedesktop.org/wiki/Software/systemd>.

Buildroot开发者推荐的方案是，对于大部分嵌入式系统来说，使用Busybox init就足够了。Systemd用在更复杂的场景。

# 7. 配置选项

在尝试修改以下任何组件之前，请确保您已经配置了Buildroot本身，并启用了相应的软件包。

## 7.1 BusyBox

如果您已经有BusyBox配置文件，则可以使用BR2\_PACKAGE\_BUSYBOX\_CONFIG直接在Buildroot配置中指定此文件。 否则，Buildroot将从默认的BusyBox配置文件开始。

如果想更新配置文件里的内容，可以使用命令make busybox-menuconfig打开BusyBox配置编辑器。

也可以使用环境变量指定BusyBox配置文件，尽管这不被推荐。详细信息可以参考[8.6节 环境变量](#_8.6_环境变量)。

## 7.2 uClibc

使用配置选项BR2\_UCLIBC\_CONFIG指定已有的配置文件。后续配置使用make uclibc-menuconfig。

## 7.3 Linux kernel

使用配置选项BR2\_LINUX\_KERNEL\_USE\_CUSTOM\_CONFIG指定已有的配置文件。后续配置使用make uclibc-menuconfig。

或者通过配置选项BR2\_LINUX\_KERNEL\_USE\_DEFCONFIG使用Buildroot默认配置文件，也可以通过配置选项BR2\_LINUX\_KERNEL\_USE\_CUSTOM\_CONFIG创建一个空文件，指定它作为用户自定义配置文件。

后续配置使用make linux-menuconfig。

## 7.4 Barebox

同linux kernel。配置变量是BR2\_TARGET\_BAREBOX\_USE\_CUSTOM\_CONFIG和BR2\_TARGET\_BAREBOX\_USE\_DEFCONFIG。

使用make barebox-menuconfig打开配置编辑器。

## 7.5 U-Boot

同linux kernel。配置变量是BR2\_TARGET\_UBOOT\_USE\_CUSTOM\_CONFIG和BR2\_TARGET\_UBOOT\_USE\_DEFCONFIG。

使用make barebox-menuconfig打开配置编辑器。

# 8. 常用Buildroot方法

## 8.1 make

（1）显示所有执行的命令

|  |
| --- |
| $ make V=1 <target> |

（2）显示有defconfig的板子，列表显示

|  |
| --- |
| $ make list-defconfigs |

（3）显示所有可用的target

|  |
| --- |
| $ make help |

并不是所有target都是可用的，.config里的一些设定项可以隐藏target：

busybox-menuconfig只有当busybox被使能时，才会工作；

linux-menuconfig和linux-savedefconfig 只有当linux 被使能时，才会工作；

uclibc-menuconfig只有当在内部工具链选择uClibc C库才能可用

barebox-menuconfig和barebox-savedefconfig只有当barebox bootloader被使能时才可用。

uboot-menuconfig和uboot-savedefconfig只有当U-Boot bootloader被使能时才可用。

Clean时，当任何体系结构或工具链配置选项发生变化时，都需要进行显式清理。

删除所有构建项目（包括构建目录，host，staging和target tree，镜像和工具链）：

|  |
| --- |
| $ make clean |

产生帮助手册：目前的帮助手册位于docs/manual目录下。为了产生帮助手册，可以使用下面的命令：

|  |
| --- |
| $ make manual-clean  $ make manual |

Outputs帮助文档将在输出/文档/手册中生成。

值得注意的是，构成这些文档的话会需要一些工具（查看[2.2 “可选的安装包](#_2.2_可选包)”）

重设buildroot以产生新的target：删除所有的构建项目以及配置：

|  |
| --- |
| $ make distclean |

注意：如果ccache被使能，运行make clean或distclean不会清空Buildroot使用的编译器cache。为了删除它，参考（“[8.12.3 在Buildroot使用ccache](#_8.12.3_在Buildroot使用ccache)”）

显示所有的内部make变量：

|  |
| --- |
| $ make -s printvars  VARIABLE=value\_of\_variable  ... |

可以使用一些变量来调整输出：

* VARS将列表限制为名称与指定make模式匹配的变量
* 如果设置为YES，QUOTED\_VARS将使用单引号扩起值
* RAW\_VARS，如果设置为YES，将打印未展开的值

例如：

|  |
| --- |
| $ make -s printvars VARS=BUSYBOX\_%DEPENDENCIES  BUSYBOX\_DEPENDENCIES=skeleton toolchain  BUSYBOX\_FINAL\_ALL\_DEPENDENCIES=skeleton toolchain  BUSYBOX\_FINAL\_DEPENDENCIES=skeleton toolchain  BUSYBOX\_FINAL\_PATCH\_DEPENDENCIES=  BUSYBOX\_RDEPENDENCIES=ncurses util-linux |

|  |
| --- |
| $ make -s printvars VARS=BUSYBOX\_%DEPENDENCIES QUOTED\_VARS=YES  BUSYBOX\_DEPENDENCIES='skeleton toolchain'  BUSYBOX\_FINAL\_ALL\_DEPENDENCIES='skeleton toolchain'  BUSYBOX\_FINAL\_DEPENDENCIES='skeleton toolchain'  BUSYBOX\_FINAL\_PATCH\_DEPENDENCIES=''  BUSYBOX\_RDEPENDENCIES='ncurses util-linux' |

|  |
| --- |
| $ make -s printvars VARS=BUSYBOX\_%DEPENDENCIES RAW\_VARS=YES  BUSYBOX\_DEPENDENCIES=skeleton toolchain  BUSYBOX\_FINAL\_ALL\_DEPENDENCIES=$(sort $(BUSYBOX\_FINAL\_DEPENDENCIES) $(BUSYBOX\_FINAL\_PATCH\_DEPENDENCIES))  BUSYBOX\_FINAL\_DEPENDENCIES=$(sort $(BUSYBOX\_DEPENDENCIES))  BUSYBOX\_FINAL\_PATCH\_DEPENDENCIES=$(sort $(BUSYBOX\_PATCH\_DEPENDENCIES))  BUSYBOX\_RDEPENDENCIES=ncurses util-linux |

引用变量的输出可以在shell脚本中重用，例如：

|  |
| --- |
| $ eval $(make -s printvars VARS=BUSYBOX\_DEPENDENCIES QUOTED\_VARS=YES)  $ echo $BUSYBOX\_DEPENDENCIES  skeleton toolchain |

## 8.2 了解何时需要全面构建

当通过make menuconfig，make xconfig或其它的配置工具修改系统配置时，Buildroot不会尝试检测系统的哪些部分应该重建。在某些情况下，Buildroot应该重建整个系统，在某些情况下，只重建一个特定的软件包子集。但是以完全可靠的方式检测这个非常困难，因此Buildroot开发人员决定不尝试这样做。

相反，用户有责任了解何时需要全面重建。下面是几条经验法则，可以帮助您了解如何使用Buildroot：

目标体系结构配置更改后，需要完整重建。例如，更改体系结构变体，二进制格式或浮点策略会影响整个系统。

当工具链配置改变时，通常需要完整的重建。更改工具链配置通常涉及更改编译器版本，C库类型或其配置或其他基本配置项目，并且这些更改会影响整个系统。

When an additional package is added to the configuration, a full rebuild is not necessarily needed. Buildroot will detect that this package has never been built, and will build it. However, if this package is a library that can optionally be used by packages that have already been built, Buildroot will not automatically rebuild those. Either you know which packages should be rebuilt, and you can rebuild them manually, or you should do a full rebuild. For example, let’s suppose you have built a system with the ctorrent package, but without openssl. Your system works, but you realize you would like to have SSL support in ctorrent, so you enable the openssl package in Buildroot configuration and restart the build. Buildroot will detect that openssl should be built and will be build it, but it will not detect that ctorrent should be rebuilt to benefit from openssl to add OpenSSL support. You will either have to do a full rebuild, or rebuild ctorrent itself.

当一个额外的软件包被添加到配置中时，不一定需要完全重建。 Buildroot会检测到这个软件包从未被构建过，并且会构建它。但是，如果此包是一个可以由已经构建的包可选使用的库，则Buildroot不会自动重建这些包。要么你知道应该重建哪些软件包，并且你可以手动重建它们，或者你应该做一个完整的重建。例如，假设您已经构建了一个包含ctorrent包的系统，但没有openssl。您的系统可以正常工作，但是您意识到您希望在ctorrent中拥有SSL支持，因此您可以在Buildroot配置中启用openssl包并重新启动构建。 Buildroot会检测到应该构建openssl并构建它，但它不会检测到应该重建crtrent以从openssl中获益以添加OpenSSL支持。你将不得不做一个完整的重建，或重建ctorrent本身。

When a package is removed from the configuration, Buildroot does not do anything special. It does not remove the files installed by this package from the target root filesystem or from the toolchain sysroot. A full rebuild is needed to get rid of this package. However, generally you don’t necessarily need this package to be removed right now: you can wait for the next lunch break to restart the build from scratch.

当一个包被从配置中移除时，Buildroot没有做任何特别的事情。它不会从目标根文件系统或工具链sysroot中删除此软件包安装的文件。完全重建需要摆脱这个包。但是，通常情况下，您不一定需要立即删除此软件包：您可以等待下一次午餐时间重新开始构建。

When the sub-options of a package are changed, the package is not automatically rebuilt. After making such changes, rebuilding only this package is often sufficient, unless enabling the package sub-option adds some features to the package that are useful for another package which has already been built. Again, Buildroot does not track when a package should be rebuilt: once a package has been built, it is never rebuilt unless explicitly told to do so.

当软件包的子选项发生更改时，软件包不会自动重建。做出这样的更改后，只重建此软件包通常就足够了，除非启用软件包子选项会在软件包中添加一些功能，这些功能对于已经构建的另一个软件包很有用。同样，Buildroot不会跟踪何时应该重建包：一旦包已经被构建，它就不会被重建，除非明确告知这样做。

When a change to the root filesystem skeleton is made, a full rebuild is needed. However, when changes to the root filesystem overlay, a post-build script or a post-image script are made, there is no need for a full rebuild: a simple make invocation will take the changes into account.

当对根文件系统骨架进行更改时，需要完整的重建。但是，如果对根文件系统覆盖，生成后脚本或后映像脚本进行更改，则不需要完全重建：简单make调用将考虑到这些更改。

Generally speaking, when you’re facing a build error and you’re unsure of the potential consequences of the configuration changes you’ve made, do a full rebuild. If you get the same build error, then you are sure that the error is not related to partial rebuilds of packages, and if this error occurs with packages from the official Buildroot, do not hesitate to report the problem! As your experience with Buildroot progresses, you will progressively learn when a full rebuild is really necessary, and you will save more and more time.

一般来说，当您面临构建错误并且您不确定所做配置更改的潜在后果时，请执行完整重建。如果你得到相同的构建错误，那么你确定这个错误与包的部分重建无关，如果这个错误发生在官方Buildroot的包中，请不要犹豫报告问题！随着您使用Buildroot的经验不断发展，您将逐步了解完全重建的真正必要性，并且您将节省越来越多的时间。

For reference, a full rebuild is achieved by running:

作为参考，完整的重建是通过下面的命令实现的：

|  |
| --- |
| $ make clean all |

## 8.3 了解如何重建软件包

Buildroot用户问的一个最常见的问题就是，如何重建一个给定的软件包或者如何删除一个软件包而不用重新构建。

## 8.4 脱机构建

## 8.5 构建输出到指定目录

## 8.6 环境变量

## 8.7高效处理文件系统映像

## 8.8 绘制包之间的依赖关系

## 8.9 绘制构建持续时间

## 8.10 绘制软件包的文件系统大小贡献

## 8.11 与Eclipse集成

## 8.12 高级用法

### 8.12.1 在Buildroot之外使用生成的工具链

### 8.12.2 在Buildroot中使用gdb

### 8.12.3 在Buildroot中使用ccache

### 8.12.4 下载的软件包的位置

### 8.12.5 打包特定的目标

运行make <package>构建和安装特定的安装包和它的依赖。

|  |
| --- |
| make <package>-<target> |

一个安装包可编译的目标都有（按照它们的执行顺序）：

|  |  |
| --- | --- |
| command/target | 描述 |
| source | 获取源代码（下载压缩包，clone源码仓库等） |
| depends | 构建和安装编译某一安装包的所有依赖 |
| extract | 把源代码放入安装包的build目录中（抽取压缩包，复制源代码等） |
| patch | 如果有的话，应用补丁 |
| configure | 运行配置命令，如果有的话 |
| build | 运行编译命令 |
| install-staging |  |
| Install-target |  |
| install |  |

# 9 添加用户自定义Linux内核驱动程序

这里我们简单的写一个hello字符驱动，添加到内核里编译。我们的内核是使用buildroot生成的，位于/home/qemu/buildroot-2012.05/output/build/linux-3.3.7/下，终端进入目录，然后添加字符驱动，首先进入到driver/char下新建一个hello\_world\_test.c，随便写一个简单的hello world字符驱动，如下：

|  |
| --- |
| /\*  \* Author: aaron shen  \* E-mail: shenwanjiang2013@163.com  \*/  #include <linux/init.h>  #include <linux/module.h>  MODULE\_LICENSE("Dual BSD/GPL");  static int hello\_init(void)  {  printk(KERN\_ALERT "Hello, world\n");  return 0;  }  static void hello\_exit(void)  {  printk(KERN\_ALERT "Goodbye, cruel world\n");  }  module\_init(hello\_init);  module\_exit(hello\_exit); |

保存后，修改当前目录下的Makefile，

添加一句：obj-m += hello\_world\_test.o

保存返回内核根目录，输入make ARCH=arm CROSS\_COMPILE=/交叉编译工具的绝对路径/arm-unknown-linux-uclibcgnueabi- modules编译，

这种方法是把所有的模块重新编译一遍。编译成功后，拷贝hello\_world\_test.ko到rootfs文件系统内，使用insmod（或modprobe）和rmmod命令查看结果：

|  |
| --- |
| # modprobe hello\_world\_test.ko  Hello, world  # rmmod hello\_world\_test.ko  Goodbye, cruel world |

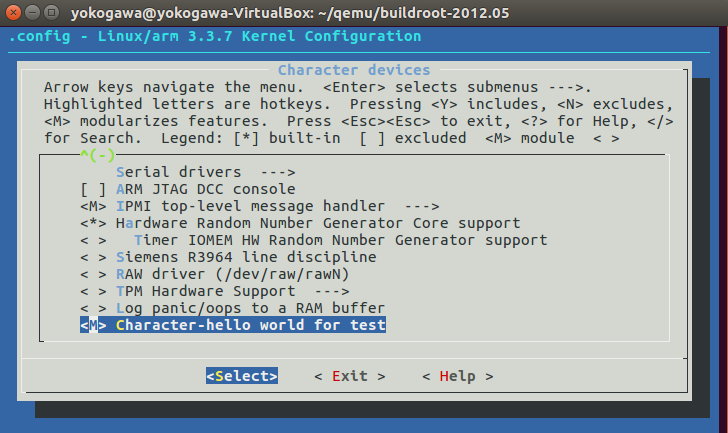
2. 也可以该模块放到整个工程中进行编译，修改hello\_world\_test.c源码所在目录的Makefile，添加一行代码：

|  |
| --- |
| obj-$(CONFIG\_HELLO\_WORLD) += hello\_world\_test.o |

再修改同一个目录下的Kconfig文件，添加：

|  |
| --- |
| config HELLO\_WORLD  tristate "Character-hello world for test"  default m  help  None. |

修改完成，保存。回到buildroot源代码根目录下，输入make linux-menuconfig，单独配置linux内核，如下所示，



可以配置为y，直接编译进内核，n不要编译这个模块，m编译为模块，系统运行后再加载。

保存配置文件，make编译即可。