OpenBricks Embedded Linux Framework - User Manual

Contents

Chapter 1

OpenBricks Introduction

1.1 What is it ?

OpenBricks is an enterprise-grade embedded Linux framework that provides easy creation of custom distributions for industrial embedded devices. It features a complete embedded development kit for rapid deployment on x86, ARM, PowerPC and MIPS systems with support for industry leaders. Pick your device, select your software bricks and cook your product !

1.2 Who is it for ?

Individuals and companies that look for rapid board bring-up with fine-grain embedded Linux distribution setup with complete customization. Ever had to care about BSP and toolchain ? That's now long gone history. If time to market means for you, OpenBricks will save your day.

1.3 Which hardware is supported ?

OpenBricks supports a broad range of embedded partners (including but not limited to Intel, TI, nVidia, Freescale, Broadcom and Marvell) and SoC, from low-end MIPS to high-end ARM Cortex-A9 MP through Intel ATOM. Whether you're designing a smartphone, a SetTopBox, a NAS or a router, OpenBricks can optimize your code for multi-cores SMP, multi-threaded SMT, hardware cryptographic accelerators, various DSPs and SIMD extensions.

1.4 What does the software offer ?

OpenBricks reduces development efforts by abstracting the low-level interface to your device. It supports all Khronos industry standards (OpenGL|ES, OpenVG, OpenMAX . . .) and major applicative frameworks (Qt, GTK+, EFL, SDL) for you to only focus on your end-user application.

1.5 Who's using it ?

OpenBricks is an OpenSource framework. It's the masterpiece framework behind your next design product. Anyone can use it and contribute to it, individuals as well as professionals. OpenBricks currently sustains the GeeXboX project.

Chapter 2

List of supported features

2.1 Key Features

- Fully Open Source and royalty-free.
- Multi-Cores SMP optimizations.
- Support for SMT HyperThreading.
- Hardware Cryptographic Acceleration: SHA1, MD5, AES ...
- Support for TouchScreens.
- Support for LIRC Infrared Remote Controls.
- Highly efficient Parallelized init.

2.2 Applicative Toolkits

- QT
- GTK
- EFL: Enlightenment Foundation Libraries
- SDL: Simple DirectMedia Layer

2.3 Graphic Extensions

- Native framebuffer Interface.
- Accelerated DirectFB engine.
- Accelerated X11 Infrastructure.
- Desktop OpenGL 3.0
- EGL Native Platform Graphics Interface
- Embedded OpenGL|ES 2.1
- Embedded OpenVG 1.0

2.4 Video Extensions

- Hardware DSP Acceleration.
- OpenMAX
- VDPAU
- VA-API

2.5 Audio Extensions

- ALSA
- PulseAudio

2.6 Media Players

- libplayer Audio/Video abstraction framework
- FFmpeg
- MPlayer
- Xine
- GStreamer
- VLC
- VDR (Video Disk Recorder)

2.7 Key Audio/Video Profiles

- Video Codecs: MPEG 1/2/4, H.264, Theora, VC-1, VP8 ...
- Audio Codecs: MP3, Vorbis, AAC, AC-3, DTS ...
- Protocols: CDDA, DVD, DVB-C/S/T, V4L2, Bluray ...
- Streaming: RTP, RTSP, ASF, MMS, WebM ...

2.8 Networking Features

- Gigabit Support
- ConnMan network manager
- WiFi with WEP and/or WPA(2) support.
- BlueTooth
- Samba Client/Server/Automounter
- NFS Client/Automounter
- Plan 9 Client/Automounter
- UPnP / DLNA support

2.9 Supported Filesystems

- EXT 2/3/4
- JBD
- ReiserFS
- JFS
- XFS
- GFS2
- OCFS2
- FUSE
- ISO9660 / Joliet / UDF
- FAT16 / FAT32 / NTFS

2.10 Toolchain Features

- Supported Target Languages: C, C++, Python
- Complete Cross-Compiler and Sysroot generation
- Support for external CodeSourcery toolchain for ARM targets
- Modularized distribution.
- Multiple supported C libraries: eglibc, glibc, uClibc
- SIMD Optimizations: NEON, VFP, AltiVec, MMX, SSE.
- Generic CPU support or target specific fine tuning optimizations.
- OPKG Packaging
- Support for extra proprietary additions (drivers, firmwares . . .).

Chapter 3

OpenBricks Supported Platforms

3.1 Supported Hardware Architectures

The OpenBricks framework aims are enabling embedded Linux distribution creation for as much hardware architectures as possible. It currently supports at least the following ones:

- ARM:
  - ARMv7 (Cortex-A9) with NEON extensions such as TI OMAP4.
  - ARMv7 (Cortex-A9) such as nVidia Tegra250.
  - ARMv7 (Cortex-A8) with NEON extensions such as TI OMAP3.
  - ARMv6 (ARM11) (with optional VFP extensions) such as Broadcom BCMring.
  - ARMv5 (ARM9) such as Marvell Kirkwood.
  - ARMv4t (ARM9TDMI) such as Samsung S3C24xx.
- x86:
  - 32 and 64 bits
  - From i586 to Core-i7.
- PowerPC
- MIPS

3.2 Available Platforms

A platform, in OpenBricks terminology, is a subset of a given hardware architecture. It can vary from a specific SoC or microprocessor brand, to a dedicated embedded board. Each platform may have some specific configuration and tuning, either hardware or software but, once certified to work with OpenBricks, it means that every owner of such a platform is ensured that the OpenBricks project will run on it.

The OpenBricks project is continuously trying to support as much platforms as possible and donating / sponsoring is the best way to have new ones being supported. The exhaustive list of supported platforms can be found **config/platform/$arch** directory.

So far, the OpenBricks project is available on the following platforms:

- ARM
  - *bcmring*: Broadcom BCM11107 boards

        * **bcm11107**: Broadcom BCM11107 evaluation board
    - *kirkwood*: All Marvell Kirkwood-equipped boards
        * **sheevaplug**: Marvell SheevaPlug
    - *omap3*: All TI OMAP3-equipped boards
        * **beagleboard**: BeagleBoard
        * **igepv2**: ISEE IGEPv2
        * **n900**: Nokia N900
        * **tao3530**: TechNexion TAO-3530 Thunder Board
        * **touchbook**: Always Innovating Touch Book
        * **zoom2**: TI OmapZoom 2
    - *omap4*: All TI OMAP4-equipped boards
        * **pandaboard**: PandaBoard
    - *s3c24xx*: All Samsung S3X24xx equipped boards
        * **gta01**: OpenMoko Neo 1973
        * **gta02**: OpenMoko Neo FreeRunner
    - *tegra2*: All nVidia Tegra250-equipped boards
        * **harmony**: nVidia Harmony evaluation board
    - *versatilepb*: ARM Versatile Platform Baseboard (ARM9 reference platform)
        * **qemu**: versatilepb machine as simulated by QEMU
    - *ux500*: ST-Ericsson U8500/U5500 smartphone platform
        * **snowball**: ST-Ericsson Snowball board
- i386
    - *generic*: 32 bits x86 i586+ PCs.
        * **vmware**: 32bits x86 VMware-optimized virtual machine.
- mips
    - *generic*: 32bits MIPS.
- powerpc
    - *generic*: 32bits PowerPC G3 Macintosh.
- powerpc64
    - *generic*: 64bits PowerPC G5 Macintosh.
- x86_64
    - *generic*: 64bits x86 AMD64 / Intel PCs.
        * **vmware**: 64bits x86 VMware-optimized virtual machine.
    - *ion*: 64bits x86 nVidia ION systems.
        * **kbb**: Koala KBB

3.3 Certified Platforms

While OpenBricks runs natively on all x86 computers, the ARM/PowerPC/MIPS embedded space is obviously another story. So far, the OpenBricks project has been tested, evaluated and certified with the following embedded platforms that we own:

- **PandaBoard (OMAP4)**
- **BeagleBoard (OMAP3)**
- **Nokia N900 (OMAP3)**
- **TI OmapZoom 2 (OMAP3)**
- **ISEE IGEPv2 (OMAP3)**
- **AlwaysInnovating TouchBook (OMAP3)**
- **nVidia Harmony (Tegra250)**
- **ST-Ericsson Snowball (U8500)**

Chapter 4

OpenBricks Toolchain Overview

A toolchain is a set of tools used to compile, link, assemble . . . source files into some kind of binary format that your processor can interpret. By definition, each type of architecture (x86, PowerPC, ARM . . .) or processor has its specific instruction set, on top of which can also be added some extra SIMD instructions.

A toolchain is meant to compile source into a code understandable by the target CPU it'll be running on. When working with embedded devices, the target architecture is often different from the one you're currently working on (referred as host). As a result, we often speak about cross-compiler suite, as the compiling tools are meant to run on host architecture and must produced target architecture-compatible binaries.

The toolchain must contain all the necessary tools, libs and headers for building bare Linux applications. It usually consists of:

- GNU Binutils
- GNU GCC
- A C library (eglibc, glibc, uClibc . . .)
- Linux kernel headers

Nearly all desktop Linux distributions are provided with their native toolchain. It's being used to compile programs for your system. Nearly none of them however comes with an available cross-toolchain.

There are many different ways to generate a cross toolchain and some specific tools exist for that purpose. All cross-toolchains are not equivalent though. They heavily vary depending on which versions of the previously mentionned tools they provide. They also can be more or less customized as to add some specific patches, fixes and optimizations for given architectures.

The OpenBricks project currently supports 2 different toolchains:

- The native **OpenBricks** toolchain.
- The external ARM toolchain from **CodeSourcery**.

The external ARM toolchain from CodeSourcery is known to be the reference toolchain for ARM architecture. It's heavily customized and optimized to provide the best performances. It is being used by multiple projects mostly because it can be deployed and installed as it and can build pretty much everything you need. It is commercially supported, updated once a year but also often uses a bit more dated versions of each tool than the one you may find on your regular desktop distribution. You may however have some issues when using different runtime system libs than the one used at build time, as provided by the toolchain.

By opposition, the native OpenBricks toolchain is mostly up-to-date and supports a much wider set of architectures. The toolchain is actually part of our sources and dynamically built depending on the configuration options you have selected. The toolchain is a bit less performance-wide than the CodeSourcery equivalent, but is much more versatile. It can also allows you to choose which C library to use, depending on your needs. It also completely matches the runtime libraries so you really won't have any runtime surprises.

The OpenBricks toolchain is set as the default toolchain. The toolchain to be used can be selected through *make menuconfig*. Go to *General settings / Toolchain settings / Toolchain* and select the one you'd like to use.

Chapter 5

OpenBricks Build Instructions

OpenBricks is meant for you to be able to create a fully customized system. It can build nearly anything you want and assemble the system of your dreams but it needs to be configured for that first.

After having fetch the OpenBricks sources, the very first thing to do is to start the configuration. You may proceed by doing the following:

make menuconfig

From there, you may want to select the kind of distribution flavour you want to build. The flavour is only a pre-defined configuration, but you can obviously completely customize it.

By *General setup* menu, you may want to configure the target architecture and platform you want to build your distribution for (e.g. *x86 ION* system or *ARM OMAP4* board). Depending on your build system, you may want to increase the concurrency make level as to use a maximum number of CPU cores to fasten the build process. If you work in a company and have multiple developers working on the same LAN, you may also want to specify a previously setup mirror server for packages tarball sources to be downloaded from instead of going on the Internet each time.

Next thing is to configure the various settings and the target images. You may want to build either a traditional rootfs, a bootable ISO for LiveCD or a network capable PXE-aware filesystem. Also possible is to configure the localization support.

Feel free after that to proceed with fine-tuning by selecting the features and packages you want to be included. Dependancies are automatically handled so that you shouldn't have to vary about the whole system's consistency.

Once satisfied, exit the configuration menu and make sure to save all of your changes. The resulting *config/options* file should now have been generated.

That's it, you're ready to build. Just *make* at shell prompt and wait for a couple hours, depending on the packages and configuration you have chosen.

For a exhaustive list of supported commands, you may have a look at the top-level *Makefile* but the main options are:

- **make** : proceed with a complete build of your distribution.
- **make clean**: clean up all build files.
- **make clean-doc**: clean up the documentation.
- **make dist**: create a tarball of all your sources, ready for distribution.
- **make distclean**: clean up all build files, documentation and external tarballs.
- **make doc**: cleanup the generated documentation files.
- **make get**: retrieve all tarballs and package sources.
- **make iso**: build a LiveCD ISO image, ready to be burned and used (mostly usefull for x86 targets only).
- **make menuconfig**: start the curses-based distribution configuration and customization menu.

- **make qemu**: runs your built-up distribution image in QEMU emulator.
- **make vmx**: build a VMware-compatible virtual machine image (only makes sense with x86 target build).
- **make vmx-play**: starts up the previously built virtual machine in VMware's vmplayer.

Chapter 6

OpenBricks Configuration System

6.1 Introduction

OpenBricks uses a Kconfig-based interface to allow the user to customize the system to suit his requirements. The configuration menu is built from several sources:

- the main Kconfig definition file (config/Kconfig.main)
- the architecture Kconfig file (config/Kconfig.arch)
- the generated Kconfig files (build/config/Kconfig.*)

The following table details the various generated Kconfig files:

Table 6.1: Generated Kconfig files

Kconfig file	**Creator script**	**Source material**	**Kconfig menu**
Kconfig.version	scripts/version2kconfig	VERSION	Version header
Kconfig.flavours	scripts/flavours2kconfig	config/flavours/*/meta	Flavour, Distribution name
Kconfig.arch	scripts/archs2kconfig	config/platforms/*/meta	General setup → Target arch
Kconfig.platform	scripts/platforms2kconfig	config/platforms/*/*/meta	General setup → Target platform
Kconfig.machine	scripts/machines2kconfig	config/platforms/*/*/machines/*/meta	General setup → Target machine
Kconfig.remote	scripts/remotes2kconfig	packages/lirc*/config/lircd*	Settings → Remote, Settings → Receiver
Kconfig.use	scripts/use2kconfig	config/use	Features
Kconfig.packages	scripts/meta2kconfig	packages/*/meta	Packages

After the user has completed the configuration selections, a .config file is created. The .config file is used by scripts/kconfig2options to create build/config/options, which is the file actually used by the rest of the OpenBricks build system (it is sourced through config/options).

6.2 Kconfig syntax

For general informations about the Kconfig syntax refer to DOCS/kconfig-language.txt. In addition to the syntax specifications, OpenBricks uses some conventions in its Kconfig files:

- hand-written Kconfig entries (e.g. TARGET_LIBC) are in uppercase, but subentries can be in lowercase (e.g. TARGET_LIBC_eglibc)
- all entries starting with the *OPT_* prefix are exported to build/config/options; for example, OPT_TOOLCHAIN_CXX=y in .config will become TOOLCHAIN_CXX=yes in build/config/options;
- all features have a USE_feature entry (e.g. USE_bluetooth)
- all packages have a PKG_package entry (e.g. PKG_MPlayer)

6.3 Configuration menu elements

6.3.1 Flavours

Flavours are defined in config/flavours, where every flavour has a subdirectory. Most settings are defined in the meta file:

- FLAVOUR_NAME
  - the name of the flavour
  - must coincide with the flavour directory name
- FLAVOUR_DISTRONAME
  - the user-visible flavour name
- FLAVOUR_DEPENDS
  - the packages the flavour requires to be installed
  - can be *all* to require all packages
  - defaults to "" (no package)
- FLAVOUR_USE
  - the features (i.e. use flags) the flavours requires to be enabled by default
  - can be *all* to require all features
  - defaults to "" (no feature)
- FLAVOUR_SHORTDESC
  - used as the short description for the flavour
  - should be one-line summary
- FLAVOUR_LONGDESC
  - used as the long description for the flavour

In addition, a flavour can define arch-specific depends using FLAVOUR_DEPENDS_$arch (e.g. FLAVOUR_DEPENDS_arm). A flavour can override the default BusyBox configuration with a busybox.conf file in its directory.

6.3.2 Architectures

Platforms are defined in config/platforms, where every architecture has a subdirectory. The arch settings are defined in a meta file:

- ARCH_NAME
  - the name of the architecture
  - must coincide with the arch directory name
- ARCH_DESC
  - the user-visible name of the architecture
  - will be displayed in the Kconfig menu
  - defaults to ARCH_NAME
- ARCH_HELP
  - the user-visible help text of the architecture
  - will be displayed in the Kconfig menu
- ARCH_SUBARCHS
  - a space-separated list of sub-architecture names (as understood by gcc)
  - defaults to "" (no subarchs)
- ARCH_CPUS
  - a space-separated list of CPU names (as understood by gcc)
  - these CPUs will be common to all the define subarchs

The ARCH_DESC and ARCH_CPUS variables can also be defined for each subarch, using ARCH_DESC_$subarch and ARCH_CPUS_

6.3.3 Platforms

Platforms are defined in config/platforms; every architecture has a subdirectory, and all platforms have a subdirectory under one of the arch subdirectories. Most settings are defined in a meta file:

- PLATFORM_NAME
  - the name of the platform
  - must coincide with the platform directory name
- PLATFORM_DESC
  - the user-visible name of the platform
  - will be displayed in the Kconfig menu
  - defaults to PLATFORM_NAME
- PLATFORM_CPU
  - the default CPU to be selected
  - defaults to "" (no selection"
- PLATFORM_DEPENDS

- the packages the platform requires to be installed
- defaults to "" (no package)

In addition, a platform can declare a list of use flags in PLATFORM_USE, to be able to specify per-useflag dependencies using PLATFORM_DEPENDS_$flag. A platform can also override the default kernel configuration with a linux.conf file, and the default bootargs (for ARM systems using u-boot) with boot.cfg.

A platform can override any package by creating a directory with the package name (i.e. config/platforms/$arch/$platform/packages/$pac
This can be used, e.g., to implement platform-specific kernels; in this case the plaform overrides the linux and linux-headers packages. Refer to the OMAP4 platform (config/platforms/arm/omap4) for an implementation example.

6.3.4 Machines

Machines are defined in the machines/ subdirectory in the platform directory; each machine has a subdirectory. Most settings are defined in the meta file:

- MACHINE_NAME
  - the name of the machine
  - must coincide with the machine directory name
- MACHINE_DESC
  - the user-visible name of the machine
  - will be displayed in the Kconfig menu
  - defaults to MACHINE_NAME
- MACHINE_CPU
  - the default CPU to be selected
  - defaults to "" (no selection"
- MACHINE_DEPENDS
  - the packages the machine requires to be installed
  - defaults to "" (no package)
- MACHINE_UBOOT_CONFIG
  - the name of the u-boot configuration to be used
  - defaults to "" (no configuration)
- MACHINE_XLOADER_CONFIG
  - the name of the X-loader configuration to be used
  - defaults to "" (no configuration)

In addition, a machine can declare a list of use flags in MACHINE_USE, to be able to specify per-useflag dependencies using MACHINE_DEPENDS_$flag. A machine can also override the default kernel configuration with a linux.conf file, and the default bootargs (for ARM systems using u-boot) with boot.cfg.

A machine can override any package by creating a directory with the package name (i.e. config/platforms/$arch/$platform/machines/$ma
This can be used, e.g., to implement machine-specific bootloaders (one example could be X-loader, which has no generic implementation and is machine-specific).

6.3.5 Remotes

Remotes are defined by LIRC configuration files in packages/lirc/config, and a Kconfig menu is created to allow the used to select the default remote and receiver to use.

6.3.6 Features

Features are defined in config/use and implemented through use flags. Packages can define package-specific use flags, but flags common to many packages or defining user-visible features are declared in config/use and exposed in the Kconfig interface in the Features menu. Each feature is defined through several variables:

- PKG_USE_NAME_$flag
  - the user-visible feature name
- PKG_USE_SECTION_$flag
  - the section the feature should be placed into
- PKG_USE_DEPENDS_$flag
  - a list of packages the feature requires to be installed
  - defaults to "" (no packages)
- PKG_USE_ARCH_$flag
  - a list of architecture the feature is defined for
  - defaults to "" (all architectures)

Features are grouped into sections, which are rendered as separate menues in the Kconfig interface. Sections are defined through several variables:

- PKG_USE_SECTION_DESC_$section
  - the user-visible section name
- PKG_USE_SECTION_KCONFIG_hwaccel
  - an optional block of Kconfig instructions which will be rendered in the section menu
  - defaults to ""

6.3.7 Packages

Packages are defined by meta files under the packages/ directory. The Packages menu in Kconfig is created by reading the meta files and grouping the packages by sections. Please refer to package-format.txt for more details on packages creation.

Chapter 7

OpenBricks Package Structure

7.1 Introduction

OpenBricks packages live under the packages/ directory in the source tree. Every package is composed of several elements:

- *meta*: package metadata information file
- *url*: where the package sources can be downloaded from (mostly obsoleted by PKG_URL field in meta)
- *need_unpack*: a shell script run before unpack stage, usually to check for missing depends
- *unpack*: a shell script run after unpack stage, usually to postprocess the source or manually unpack packages which require special handling
- *build*: a shell script which takes care of building the package from source
- *install*: a shell script run to construct the runtime opkg package
- *installdev*: a shell script run to construct the devel opkg package
- *config/*: a directory for misc configuration files used by the package
- *unit/*: a directory for systemd units
- *package/*: a directory for custom opkg control files (e.g. prerm scripts)
- *patches/*: a directory containing patches (in `diff -Naur` format) to be applied after unpack and before build
- *scripts/*: a directory for misc shell scripts used by the package
- *sources/*: a directory for package sources, which will be copied verbatim to the package build directory at unpack stage
- *tmpfiles.d/*: a directory for systemd-tmpfiles configuration files
- *modules-load.d/*: a directory for configuration files to list the kernel modules to load at boot
- *modprobe.d/*: a directory for kernel module options configuration files

Most packages only use a small subset of these elements — runtime programs usually have just *meta*, *build* and *install*, while libraries tend to also use *installdev* for includes and such.

7.2 Meta File Format

The *meta* file is used to provide the package metadata information (hence the name). It is a POSIX shell script which is sourced by the build system; it can contain variable assignments and conditional instructions.

A devel-only package (e.g. gmp, gcc-core) has three mandatory fields:

- PKG_NAME
  - the name of the package
  - must coincide with the package directory name
  - can be different from the upstream tarball name
  - will be used to create the package build directory
- PKG_VERSION
  - the upstream package current version
  - will be used to create the package build directory
- PKG_REV
  - the OpenBricks package revision
  - incremented on every major change in the package
  - reset to 1 every time PKG_VERSION changes

A devel-only package has several optional fields:

- PKG_URL
  - a space-separated list of URLs the package should be downloaded from
  - can start with variable *$DISTRO_SRCS* if the files are hosted on the OpenBricks server, or with *$SFNET_SRCS* if they are on SourceForge
  - in addition, URLs can be listed in the url file in the package directory; all resources listed will be fetched
  - PKG_URL can point to the location of a source code repository such as git, Subversion (svn) or Mercurial (hg) when used together with PKG_URL_PROTO; details are given in the section "Repository Download" below
- PKG_SHA256
  - a space-separated list of SHA-256 checksums of the package files
  - the checksums will be checked against the downloaded files
- PKG_MD5
  - a space-separated list of MD5 checksums of the package files
  - the checksums will be checked against the downloaded files
- PKG_BUILD_DEPENDS
  - the build time package dependencies, i.e. the packages required to be able to build the package
  - will be packaged and installed to sysroot before the package is built
  - defaults to "" (no build depends)
- PKG_DEV_DEPENDS
  - the packages required to use the dev package (e.g. gcc-core needs binutils to work, but requires gmp only to build)

  - will be installed to sysroot before the dev package is installed
  - defaults to "$PKG_BUILD_DEPENDS"

A runtime package has the same mandatory fields of a devel-only package, plus:

- PKG_RUN_DEPENDS
  - the runtime package dependencies, i.e. the packages required to be able to run the package
  - will be packaged and installed to the target system before the package is installed
  - defaults to "" (no runtime depends)
- PKG_USE
  - a list of "use flags" the package can handle
  - see section "USE FLAGS" for more details
  - defaults to "" (no use flags)
- PKG_REQUIRES_USE
  - a list of "use flags" the package requires to be enabled
  - you can also specify per-package useflags with the "package:flag" syntax
  - see section "USE FLAGS" for more details
  - defaults to "" (no required use flags)
- PKG_PRIORITY
  - the package priority, i.e. how much the package is important
  - can be
    - *required*: the system will not boot without this package (e.g. linux)
    - *standard*: this package is part of the OpenBricks base system
    - *optional*: normal priority for packages not part of OpenBricks base system
    - *extra*: this is a minor non-essential package
- PKG_SECTION
  - the package category, used to group packages by function
  - can be:
    - *admin*: tools and program useful for system administration
    - *drivers*: kernel or userspace drivers for hardware
    - *filesystem*: filesystem support drivers and programs
    - *games*: leisure programs
    - *libs*: shared libraries used by other programs
    - *multimedia*: programs dealing with audio, video or image contents
    - *net*: network clients, servers, and generic network-related programs
    - *perl*: Perl modules
    - *python*: Python modules
    - *sound*: programs dealing with audio support
    - *system*: essential system programs and libraries
    - *utils*: miscellaeous utility programs
    - *x11*: programs and libraries related to the Xorg windowing system
- PKG_SHORTDESC

  - used as the short description for the package
  - should be one-line summary
  - should not start with the package name

- PKG_LONGDESC
  - used as the long description for the package

A regular package has the same optional fields of a devel-only package, plus:

- PKG_DEPENDS
  - the package dependencies required both at build and at runtime
  - shorthand for adding a package to both PKG_RUN_DEPENDS and PKG_BUILD_DEPENDS
  - defaults to "" (no depends)
- PKG_ARCH
  - the target architectures supported by the package
  - can be:
    * *any*: the package is supported on all the available architectures
    * *all*: this is an architecture-independent package (e.g. enna-theme)
    * a spaced list of architecture names
    * defaults to "any"
- PKG_LICENSE
  - the upstream package license
  - can be:
    * *free*: the package is distributed under a DFSG-compliant license (i.e GPL, LGPL, MIT, etc.)
    * *non-free*: the package license does not meet the DFSG
    * a non-free package may restrict the distribution of the entire distro if it is built in, and may have unreasonable/difficult to meet restrictions
    * defaults to "free"

7.2.1 Use Flags

In the context of a OpenBricks package, a use flag represents a conditional feature, which can be selected by the user at compile time and which could bring alongside additional depends. Use flags are strictly per-package: flags enabled for package X do not affect flags for package Y. Some flags (e.g. xorg) may be enabled or disabled distro-wide with a config option, but their value can still be customized for each package.

A package declares its available flags with PKG_USE in meta. For each flag, a package can declare several information (all optional), using the following per-flag variables:

- PKG_USE_NAME_$flag
  - the displayed name of the use flag
  - defaults to "$flag"
- PKG_USE_DESC_$flag
  - the short description of the flag
  - defaults to "Enable $PKG_USE_NAME_$flag support"

- PKG_USE_HELP_$flag
  - the long description of the flag, which is used as its help text in the configuration menu
  - defaults to "$PKG_USE_DESC_$flag."
- PKG_USE_DEFAULT_$flag
  - the default status of the flag, i.e. if it's to be enabled or disabled
  - note that a flag will automatically default to enabled status if the option USE_$flag is enabled (this is used to globally toggle a flag status)
  - can be "yes" or "no", defaults to "no"

In addition, a flag can declare additional depends, which will be carried by the package if the flag is enabled:

- PKG_DEPENDS_$flag
- PKG_BUILD_DEPENDS_$flag
- PKG_RUN_DEPENDS_$flag

A package can also force specific flags to be enabled by declaring them in PKG_REQUIRES_USE. If the package requires a specific use flag to be enabled only for a given package, use the syntax "$package:$flag" in PKG_REQUIRES_USE. In this case, the package must also be listed in the PKG_DEPENDS. For example, PKG_REQUIRES_USE="qt:mysql xorg" forces the "xorg" flag to be globally enabled, and the "mysql" flag to be enabled for the "qt" package.

7.2.2 Repository Download

PKG_URL can point to the location of a source code repository. In this case, PKG_URL_PROTO specifies the repository type:

- git for a git repository
- hg for a Mercurial repository
- svn for a Subversion repository

Note that PKG_URL can contain any URL permissible by the respective source code management tool (such as git://, svn:// or http://).

PKG_URL_REV can optionally specify the revision/branch/tag of the source code to download. If omitted, the default source code version is obtained. It is recommended to specify a particular revision and to include PKG_URL_REV in PKG_VERSION, such as shown in the example below. Note that PKG_VERSION should start with a number or the letter *r* so that build directories can be automatically deleted.

When using PKG_URL_PROTO, only a single URL must be given.

Example

```
PKG_URL_PROTO=git
PKG_URL="git://kernel.ubuntu.com/ubuntu/ubuntu-natty.git"
PKG_URL_REV=cefe94dc5d171940edd23081d9d481dc1ed5824b
PKG_VERSION=2.6.38-natty-${PKG_URL_REV}
```

7.2.3 Subpackages

A subpackage is a special kind of runtime package, which packages an optional component of another package and is created from its install tree. A subpackage has an additional mandatory field in meta:

- PKG_PARENT
  - the name of the parent package, i.e. the package this subpackage should be built from
  - defaults to "" (empty)

A subpackage has two additional optional fields in meta:

- PKG_PARENT_USE
  - a list of use flags of the parent package
  - the subpackage will be selectable in Kconfig only if all the use flags specified are active
  - defaults to "" (no use flags)
- PKG_NO_PARENT_DEPENDS
  - whether the package should automatically build-depend on its parent or not
  - boolean field, defaults to *yes*

For clarity, it is recommended (but not required) to name a subpackage as "$PKG_PARENT-subpackagename".

7.2.4 Meta Examples

A devel-only package

```
PKG_NAME=gmp
PKG_VERSION=5.0.5
PKG_URL="http://ftp.gnu.org/gnu/${PKG_NAME}/${PKG_NAME}-${PKG_VERSION}.tar.xz"
PKG_REV=1
PKG_PRIORITY=optional
PKG_SECTION=libs
PKG_BUILD_DEPENDS="toolchain"
PKG_SHORTDESC="The GNU Multiple Precision Arithmetic Library"
PKG_LONGDESC="GMP is a free library for arbitrary precision arithmetic, operating on signed ↩
     integers, rational numbers, and floating point numbers. There is no practical limit to  ↩
    the precision except the ones implied by the available memory in the machine GMP runs on ↩
    . GMP has a rich set of functions, and the functions have a regular interface."
```

A standard runtime package

```
PKG_NAME=lsof
PKG_VERSION=4.83
PKG_URL="http://ftp.cerias.purdue.edu/pub/tools/unix/sysutils/lsof/lsof_${PKG_VERSION}.tar. ↩
    bz2"
PKG_REV=1
PKG_RUN_DEPENDS="$TARGET_LIBC"
PKG_BUILD_DEPENDS="toolchain"
PKG_PRIORITY=optional
PKG_SECTION=utils
PKG_SHORTDESC="List open files"
PKG_LONGDESC="Lsof is a Unix-specific diagnostic tool.  Its name stands for LiSt Open Files ↩
    , and it does just that.  It lists information about any files that are open, by  ↩
    processes currently running on the system."
```

Note the use of $TARGET_LIBC in PKG_DEPENDS to refer to the runtime system libc (which could be uClibc, glibc or eglibc)

A non-free package available only on selected archs

```
PKG_NAME=xf86-video-nvidia
PKG_VERSION="295.53"
[ "$TARGET_ARCH" = i386 ] && \
  PKG_URL="ftp://download.nvidia.com/XFree86/Linux-x86/${PKG_VERSION}/NVIDIA-Linux-x86-${ ↩
      PKG_VERSION}.run"
[ "$TARGET_ARCH" = x86_64 ] && \
  PKG_URL="ftp://download.nvidia.com/XFree86/Linux-x86_64/${PKG_VERSION}/NVIDIA-Linux- ↩
      x86_64-${PKG_VERSION}-no-compat32.run"
PKG_REV=1
PKG_ARCH="i386 x86_64"
PKG_LICENSE=non-free
PKG_DEPENDS="xorg-server"
PKG_BUILD_DEPENDS="toolchain"
PKG_RUN_DEPENDS="$TARGET_LIBC kmod"
PKG_USE="vdpau"
PKG_PRIORITY=optional
PKG_SECTION=x11
PKG_SHORTDESC="NVIDIA binary Xorg driver"
PKG_LONGDESC="These binary drivers provide optimized hardware acceleration of OpenGL  ↩
    applications via a direct-rendering X Server. AGP, PCIe, SLI, TV-out and flat panel  ↩
    displays are also supported. This version only supports GeForce 6xxx and higher of the  ↩
    Geforce GPUs plus complimentary Quadros and nforce."
```

A complex package which shows the usage of conditionals to define the depends

```
PKG_NAME=MPlayer
PKG_VERSION=1.1
PKG_URL="http://www.mplayerhq.hu/MPlayer/releases/MPlayer-${PKG_VERSION}.tar.xz"
PKG_REV=1
PKG_DEPENDS="zlib ffmpeg freetype alsa-lib fribidi libcdio faad2 libpng enca libass  ↩
    fontconfig libvorbis libtheora libmad"
PKG_BUILD_DEPENDS="toolchain yasm"
PKG_RUN_DEPENDS="$TARGET_LIBC"

PKG_USE="uclibc vdpau sdl xorg unrar live dvd bluray dvb"
PKG_DEPENDS_uclibc="libiconv"
PKG_DEPENDS_vdpau="libvdpau"
PKG_DEPENDS_sdl="SDL"
PKG_DEPENDS_xorg="libX11 libXv libXxf86vm"
PKG_RUN_DEPENDS_unrar="unrar"
PKG_BUILD_DEPENDS_live="live"
PKG_DEPENDS_dvd="libdvdread libdvdnav"
PKG_DEPENDS_bluray="libbluray"

PKG_PRIORITY=standard
PKG_SECTION=multimedia
PKG_SHORTDESC="movie player for Unix-like systems"
PKG_LONGDESC="MPlayer plays most MPEG, VOB, AVI, Ogg/OGM, VIVO, ASF/WMA/WMV, QT/MOV/MP4,  ↩
    FLI, RM, NuppelVideo, yuv4mpeg, FILM, RoQ, PVA files, supported by many native, XAnim,  ↩
    RealPlayer, and Win32 DLL codecs. It can also play VideoCD, SVCD, DVD, 3ivx, RealMedia,  ↩
    and DivX movies. Another big feature of MPlayer is the wide range of supported output  ↩
    drivers. It works with X11, Xv, DGA, OpenGL, SVGAlib, fbdev, DirectFB, but also SDL ( ↩
    plus all its drivers) and some low level card-specific drivers (for Matrox, 3Dfx and  ↩
    Radeon, Mach64 and Permedia3). Most of them support software or hardware scaling,  ↩
    therefore allowing fullscreen display. MPlayer is also able to use some hardware MPEG  ↩
    decoder boards, such as the DVB and DXR3/Hollywood+."
```

7.3 Package scripts

All the scripts in the package directory follow some conventions:

- the script should be written for the Bourne shell (`#!/bin/sh`); bashisms should be avoided, checking the scripts with a modern *sh* implementation such as *dash* is recommended
- the script should source *config/options* as their first action; because of this, the script will have access to all the variables and functions defined in *config/options*, *config/toolchain*, *config/path* and *config/functions*
- scripts are expected to be called with the package name as the first argument —this means `$1` inside the script will refer to the package name
- the script can use `get_meta $1` to read the package meta if necessary; directly sourcing the meta file is not recommended

7.3.1 *need_unpack* script

The *need_unpack* script is executed just before the package unpack stage. It can be used to ensure the prerequisites for the package are always satisfied. It is normally used only for packages that require a kernel tree to build (such as out-of-tree kernel drivers), to make sure they are properly rebuilt if the kernel tree changes (e.g. because of a kernel upgrade). This is obtained by removing the unpack stamp for the package if the conditions are not satisfied, which forces a full rebuild.

7.3.2 *unpack* script

The *unpack* script is executed just after the package unpack stage. It can be used to postprocess the package sources (e.g. fixing a broken upstream makefile with `sed`) or to manually unpack the package sources (e.g. for binary packages like xf86-video-nvidia). If necessary, the helper function `apply_patch <package> <patch>` can be used to apply arbitrary patches to the package. The package unpack directory can be referenced reading *$PKG_BUILD_DIR* after a *get_meta* call; for most packages where the standard unpack works it is also possible to use *$BUILD/$1**, though it could lead to conflicts in case of multiple packages with similar names.

7.3.3 *build* script

The *build* script is responsible for the package build, which for most packages entails a compilation from source. The script can be handwritten, but there are several helper functions (`do_configure`, `make_install`, etc.) which can help to automate the most boring parts. The compilation results should be installed in *$PKG_BUILD_DIR/.install*, to ease the following stages implementation.

Example

```
#!/bin/sh

. config/options

cd $BUILD/$1*

do_configure

make
make_install
```

7.3.4 *install* script

The *install* script gathers the files which should be copied to the target rootfs, which is set as *$INSTALL*. If the *build* script has created a *.install* directory it can be easily written using the `do_install` function.

Example

```
#!/bin/sh

. config/options

cd $BUILD/$1*

do_install usr/bin
do_install usr/lib/lib*.so*
```

7.3.5 *installdev* script

The *installdev* script gathers the files which should be copied to the toolchain directory, which is set as *$INSTALL*. If the *build* script has created a *.install* directory it can be easily written using the `do_installdev` function.

Example

```
#!/bin/sh

. config/options

cd $BUILD/$1*

do_installdev usr/include
do_installdev usr/lib
```

7.3.6 Helper functions

setup_toolchain <target | host> sets several environment variables (such as $CFLAGS) to prepare for a host build (using $HOST_CC) or for a target build (using $TARGET_CC). `setup_toolchain target` is automatically called before a package build, so it's not necessary to explicitly use for normal package builds.

get_meta <package> retrieves a package meta file and sources it into the environment, doing several safety checks, some postprocessing and setting additional variables.

pkg_uses <package> <use_flag> returns true if the specified use flag is currently enabled for the package, and false otherwise

kernel_path returns the path to the kernel build tree, which can be useful to build out-of-tree kernel modules

kernel_version returns the kernel version

require_glibc <package> aborts build if TARGET_LIBC is not a glibc variant

require_cxx <package> aborts build if C++ support is not enabled

do_qmake invokes qmake with the correct setup for a cross build

do_strip [bin | shlib | staticlib] <path> strips the argument, after checking that it is actually an ELF object file; the first optional argument, which defaults to *bin*, sets the correct strip options for the target

extract_debug_info <debug_path> <unstripped_files...> uses objdump to create detached debug symbols in *debug_path* for the specified files

strip_libs <path> [debug_path] calls `do_strip shlib`; if debug_path is set, also calls `extract_debug_info`

strip_bins <path> [debug_path] calls `do_strip bin`; if debug_path is set, also calls `extract_debug_info`

xorg_drv_configure_prepend fixes the include files for Xorg drivers

fix_libs <path> [toolchain | sysroot | libprefix] rewrites the prefix path in pkgconfig and library files for the specified target, which defaults to `libprefix`

make_install [toolchain | sysroot | libprefix] [unstripped] is an automagic function to ease the installation of packages using autotools: runs `make install` using *$PKG_BUILD_DIR/.install* as install prefix, calls `fix_libs` to set the correct prefix paths for libraries, calls `strip_libs` and `strip_bins` to strip ELF files and place detached debug symbols in *$PKG_BUILD_DIR/.install-debuginfo*

do_configure [host | target] [configure_options...] is an automagic function to ease the configuration of packages using autotools; it sets up the enviroment for a host or target build and calls *./configure* with the correct arguments

do_install <file> is used in the package *install* script to copy to *$INSTALL* the specified file (which can include globbing) from *$PKG_BUILD_DIR/.install*

do_installdev <file> [toolchain | sysroot | libprefix] is used in the package *installdev* script to copy to the requested target the specified file (which can include globbing) from *$PKG_BUILD_DIR/.install*

Appendix A

Understanding Kconfig File Format

A.1 Introduction

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

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

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

A.2 Menu entries

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

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

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

A.3 Menu attributes

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

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

```
bool "Networking support"
```

and

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

- input prompt: "prompt" <prompt> ["if" <expr>] Every menu entry can have at most one prompt, which is used to display to the user. Optionally dependencies only for this prompt can be added with "if".
- default value: "default" <expr> ["if" <expr>] A config option can have any number of default values. If multiple default values are visible, only the first defined one is active. Default values are not limited to the menu entry where they are defined. This means the default can be defined somewhere else or be overridden by an earlier definition. The default value is only assigned to the config symbol if no other value was set by the user (via the input prompt above). If an input prompt is visible the default value is presented to the user and can be overridden by him. Optionally, dependencies only for this default value can be added with "if".
- type definition + default value:

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

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

- dependencies: "depends on" <expr> This defines a dependency for this menu entry. If multiple dependencies are defined, they are connected with *&&*. Dependencies are applied to all other options within this menu entry (which also accept an "if" expression), so these two examples are equivalent:

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

and

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

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

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

- numerical ranges: "range" <symbol> <symbol> ["if" <expr>] This allows to limit the range of possible input values for int and hex symbols. The user can only input a value which is larger than or equal to the first symbol and smaller than or equal to the second symbol.
- help text: "help" or "---help---" This defines a help text. The end of the help text is determined by the indentation level, this means it ends at the first line which has a smaller indentation than the first line of the help text. "---help---" and "help" do not differ in behaviour, "---help---" is used to help visually separate configuration logic from help within the file as an aid to developers.
- misc options: "option" <symbol>[=<value>] Various less common options can be defined via this option syntax, which can modify the behaviour of the menu entry and its config symbol. These options are currently possible:
  - "defconfig_list" This declares a list of default entries which can be used when looking for the default configuration (which is used when the main .config doesn't exists yet.)
  - "modules" This declares the symbol to be used as the MODULES symbol, which enables the third modular state for all config symbols.
  - "env"=<value> This imports the environment variable into Kconfig. It behaves like a default, except that the value comes from the environment, this also means that the behaviour when mixing it with normal defaults is undefined at this point. The symbol is currently not exported back to the build environment (if this is desired, it can be done via another symbol).

A.4 Menu dependencies

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

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

Expressions are listed in decreasing order of precedence.

1. Convert the symbol into an expression. Boolean and tristate symbols are simply converted into the respective expression values. All other symbol types result in *n*.
2. If the values of both symbols are equal, it returns *y*, otherwise *n*.
3. If the values of both symbols are equal, it returns *n*, otherwise *y*.
4. Returns the value of the expression. Used to override precedence.
5. Returns the result of (2-/expr/).
6. Returns the result of min(/expr/, /expr/).
7. Returns the result of max(/expr/, /expr/).

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

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

A.5 Menu structure

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

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

config NETDEVICES
        ...

endmenu
```

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

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

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

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

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

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

MODVERSIONS directly depends on MODULES, this means it's only visible if MODULES is different from *n*. The comment on the other hand is always visible when MODULES is visible (the (empty) dependency of MODULES is also part of the comment dependencies).

A.6 Kconfig syntax

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

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

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

config

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

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

menuconfig

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

This is similar to the simple config entry above, but it also gives a hint to front ends, that all suboptions should be displayed as a separate list of options.

choices

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

This defines a choice group and accepts any of the above attributes as options. A choice can only be of type bool or tristate, while a boolean choice only allows a single config entry to be selected, a tristate choice also allows any number of config entries to be set to *m*. This can be used if multiple drivers for a single hardware exists and only a single driver can be compiled/loaded into the kernel, but all drivers can be compiled as modules. A choice accepts another option "optional", which allows to set the choice to *n* and no entry needs to be selected.

comment

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

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

menu

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

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

if

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

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

source

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

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

mainmenu

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

This sets the config program’s title bar if the config program chooses to use it.

A.7 Kconfig hints

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

A.7.1 Adding common features and make the usage configurable

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

We would in lib/Kconfig see:

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

config GENERIC_IOMAP
        depends on HAVE_GENERIC_IOMAP && FOO
```

And in lib/Makefile we would see:

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

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

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

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

Note
the use of the internal config variable HAVE_GENERIC_IOMAP, it is introduced to overcome the limitation of select which will force a config option to *y* no matter the dependencies. The dependencies are moved to the symbol GENERIC_IOMAP and we avoid the situation where select forces a symbol equals to *y*.

A.7.2 Build as module only

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

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

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

Appendix B

Adding a new package

Packages are defined in packages/$package. To easily add a new package you can use the *newpackage* tool, which will create a skeleton package template for you to fill in. Simply run ./scripts/newpackage hello to create a package for the *hello* program. The script will create a new *hello* directory under packages. To quickly package your software:

1. fill in the meta file fields, starting from PKG_VERSION and PKG_URL;
2. run *./scripts/unpack hello* to make sure package download is ok, and check for the presence of *hello-$PKG_VERSION* directory in $BUILD;
3. check packages/hello/build, and add additional configure options to the do_configure call; for packages not using GNU autotools you may have to do more extensive customizations;
4. run *./scripts/build hello* and make sure the package builds with no errors;
5. check $BUILD/hello-$PKG_VERSION/.install to see which programs/libraries should be installed in the target system;
6. edit packages/hello/install to select the files to install to the target system;
7. if you are packaging a library, edit packages/hello/installdev to select the files that should be installed in toolchain; otherwise remove the file;
8. run *./scripts/clean hello* to clean the package
9. try to select the package in the configuration system and build an image with it

Appendix C

Adding a new architecture

Architectures are defined in config/plaform. To add support for a new architecture, start by creating a subdirectory with you arch name and edit the meta file, using one of the existing architectures as example. After defining the arch, you should also add generic platform and machine definitions.

Appendix D

Adding a new platform

Platforms are defined in config/plaform/$arch/$platform. To add support for a new platform, you should copy the *generic* platform for your arch: cp -PR config/plaform/myarch/generic config/plaform/myarch/myplatform and edit meta to change the platform settings.

At this point you have to decide if you want to use the OpenBricks kernel or a vendor-supplied kernel. In the first case, you can add arch-specific patches to packages/linux/patches or, if they break other archs/platforms, to config/plaform/myarch/myplatform/packages/linux/patches. You should also add a tuned kernel config for your platform as linux.conf in the platform directory.

If you want to use a vendor kernel, you need to override the *linux* and *linux-headers* packages. Create config/plaform/myarch/myplatform/packages/linux/meta with PKG_VERSION and PKG_URL pointing to your kernel, and config/plaform/myarch/myplatform/pa
headers/meta with a matching PKG_VERSION.

If your platform uses u-boot, you can add platform-specific bootargs to config/plaform/myarch/myplatform/boot.cfg.

Appendix E

Adding a new machine

Machines are defined per-platform in config/plaform/$arch/$platform/machines. To add support for a new machine, you should copy an existing machine definition (such as the *generic* machine for your arch): cp -PR config/plaform/myarch/myplatform/machines/generic config/plaform/myarch/myplatform/machines and edit meta to change the machine settings.

At this point you have to decide if you want to use the OpenBricks kernel or a vendor-supplied kernel. In the first case, you can add arch-specific patches to packages/linux/patches or, if they break other archs/platforms, to config/plaform/myarch/myplatform/packages/linux/patches. You should also add a tuned kernel config for your platform as linux.conf in the platform directory.

If you want to use a vendor kernel, you need to override the *linux* and *linux-headers* packages. Create config/plaform/myarch/myplatform/packages/linux/meta with PKG_VERSION and PKG_URL pointing to your kernel, and config/plaform/myarch/myplatform/pa
headers/meta with a matching PKG_VERSION.

If your machine uses u-boot, you can add machine-specific bootargs to config/plaform/myarch/myplatform/machines/mymachine/boot.cfg.

Appendix F

Adding a new distribution flavour

Distribution flavours are defined in config/flavours/$flavour. To add a new flavour *example* you can follow these instructions:

1. mkdir config/flavours/example
2. create config/flavours/example/meta with a text editor, with the following contents:

```
FLAVOUR_NAME=example
FLAVOUR_DISTRONAME="Example Flavour"
FLAVOUR_DEPENDS=""
FLAVOUR_USE=""
FLAVOUR_SHORTDESC="my example flavour"
FLAVOUR_LONGDESC="a detailed description of my example flavour"
```

1. add the list of packages you want to select in your flavour to FLAVOUR_DEPENDS; have a look at the existing flavours (especially *base*, which is a minimal console-only system) for examples of package combinations
2. optionally, add arch-specific packages to FLAVOUR_DEPENDS_$arch
3. optionally, add platform-specific packages to FLAVOUR_DEPENDS_$arch_$platform
4. add the list of features (i.e. use flags) you want to enable to FLAVOUR_USE
5. optionally, add a BusyBox configuration file as busybox.conf to customize BusyBox configuration

After creating the meta file, the new flavour will be available for selection in the OpenBricks configuration system. If you believe your flavour can be of general usage we encourage you to submit it to the OpenBricks mailing list for review and inclusion in the upstream tree.