The OpenADK user manual

Contents

OpenADK usage and documentation by Waldemar Brodkorb.

(based on the buildroot manual by Thomas Petazzoni. Contributions from Karsten Kruse, Ned Ludd, Martin Herren and others. See http://www.buildroot.net for the original text).

Chapter 1

About OpenADK

OpenADK is a tool that simplifies and automates the process of building a complete Linux system for an embedded system, using cross-compilation. ADK stands for appliance development kit.

In order to achieve this, OpenADK is able to generate a cross-compilation toolchain, a root filesystem, a Linux kernel image and a bootloader for your target.

OpenADK is useful mainly for people working with embedded systems, but can be used by people playing with emulators or small netbooks needing a fast and small Linux system.

OpenADK can also be used to generate a cross-toolchain for any kind of architecture and C library combination. It supports uClibc-ng, musl, GNU libc and newlib. With newlib support you can build bare-metal toolchains without need for Linux as operating system.

Embedded systems often use processors that are not the regular x86 processors everyone is used to having in his PC.

OpenADK supports 40 different architectures: AARCH64, Alpha, ARC, ARM, AVR32, Blackfin, C6X, CR16, CRIS, Epiphany, FR-V, H8/300, HPPA, IA64, LM32, M32R, M68K, METAG, Microblaze, MIPS, MIPS64, MN10300, Moxie, MSP430, NDS32, NIOS2, OR1K, PPC, PPC64, RiscV, RX, S/390, SH, SPARC, SPARC64, Tile, V850, X86, X86_64 and Xtensa.

OpenADK supports numerous processors and their variants; it also comes with sample configurations for many embedded systems, emulators and netbooks.

OpenADK is not a Linux distribution and there are no releases or binary packages available. If you need something like that, better switch to something else. OpenADK builds everything from source. There are only a few exceptions to this rule (f.e. some bootloaders and firmware files for wireless network cards).

Chapter 2

Starting up

2.1 System requirements

OpenADK is designed to run on Linux systems. But there is support to run on MacOS X, Windows with Cygwin, OpenBSD, MirBSD, NetBSD and FreeBSD. Main development happens on Debian/GNU Linux and MacOS X. The other host platforms are occasionally tested. OpenADK detects the host system and displays only the software packages, which are known to be cross-compilable on the used host. For example OpenJDK7 is only cross-compilable on a Linux host.

OpenADK needs some software to be already installed on the host system; here is the list of the mandatory packages, package names may vary between host systems.

- Build tools:
  - `binutils`
  - `C compiler (gcc or clang)`
  - `C++ compiler (g++ or clang++)`
  - `make`
  - `gzip`
  - `perl`
  - `tar`
  - `git`
  - `strings`
  - `curl or wget`
  - `ncurses development files`
  - `zlib development files`
  - `libc development files`

There is a check for the required versions of these tools in advance, though.

For some packages there are some optional packages required. OpenADK will check for the required tools in advance, when a specific package is choosen. For example Kodi needs Java installed on the host system. OpenADK tries to avoid any optional required host tools and will try to build them when needed.

2.2 Getting OpenADK

OpenADK does not have any releases. We are following the rolling release development model.

To download OpenADK using Git just do:

```
$ git clone git://openadk.org/git/openadk
```

Or if you prefer HTTP or using Git behind a proxy:

```
$ git clone http://git.openadk.org/openadk.git
```

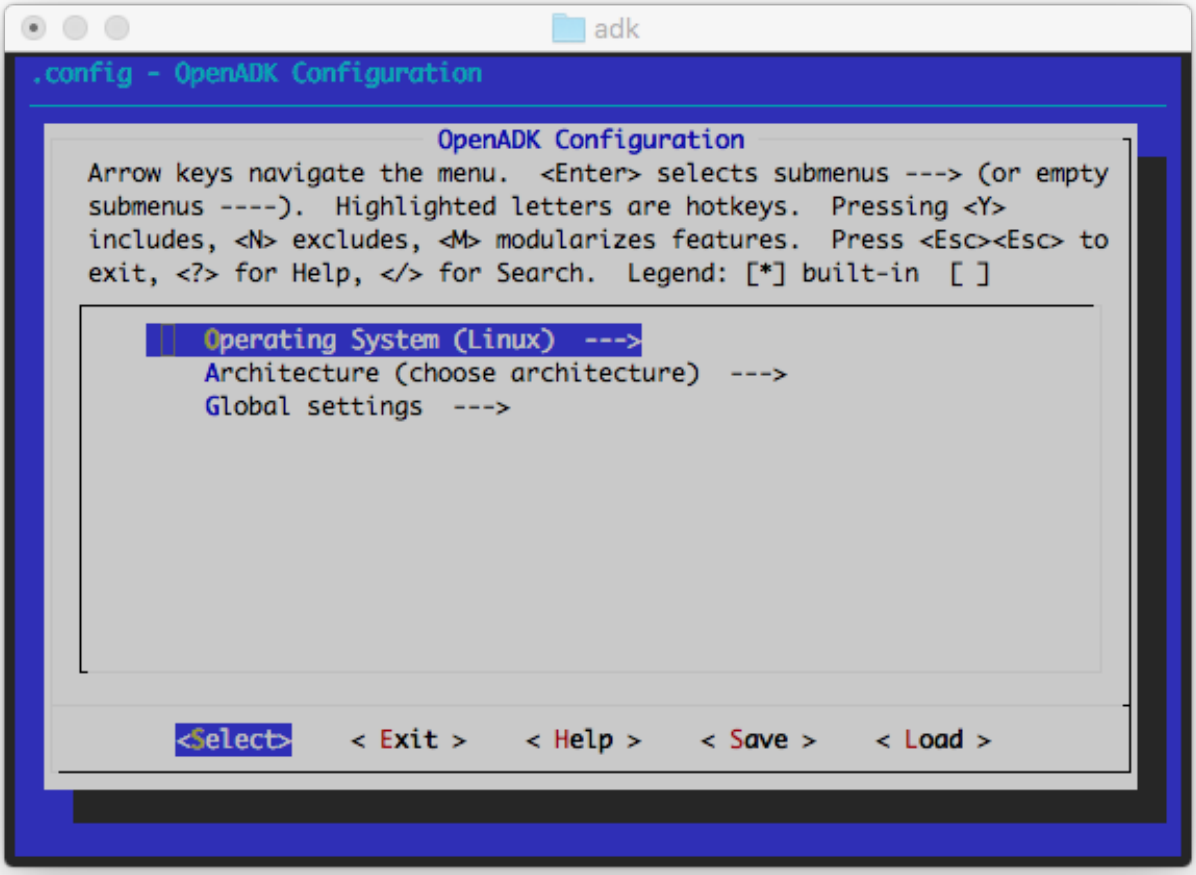
Or you can get a snapshot.

2.3 Using OpenADK

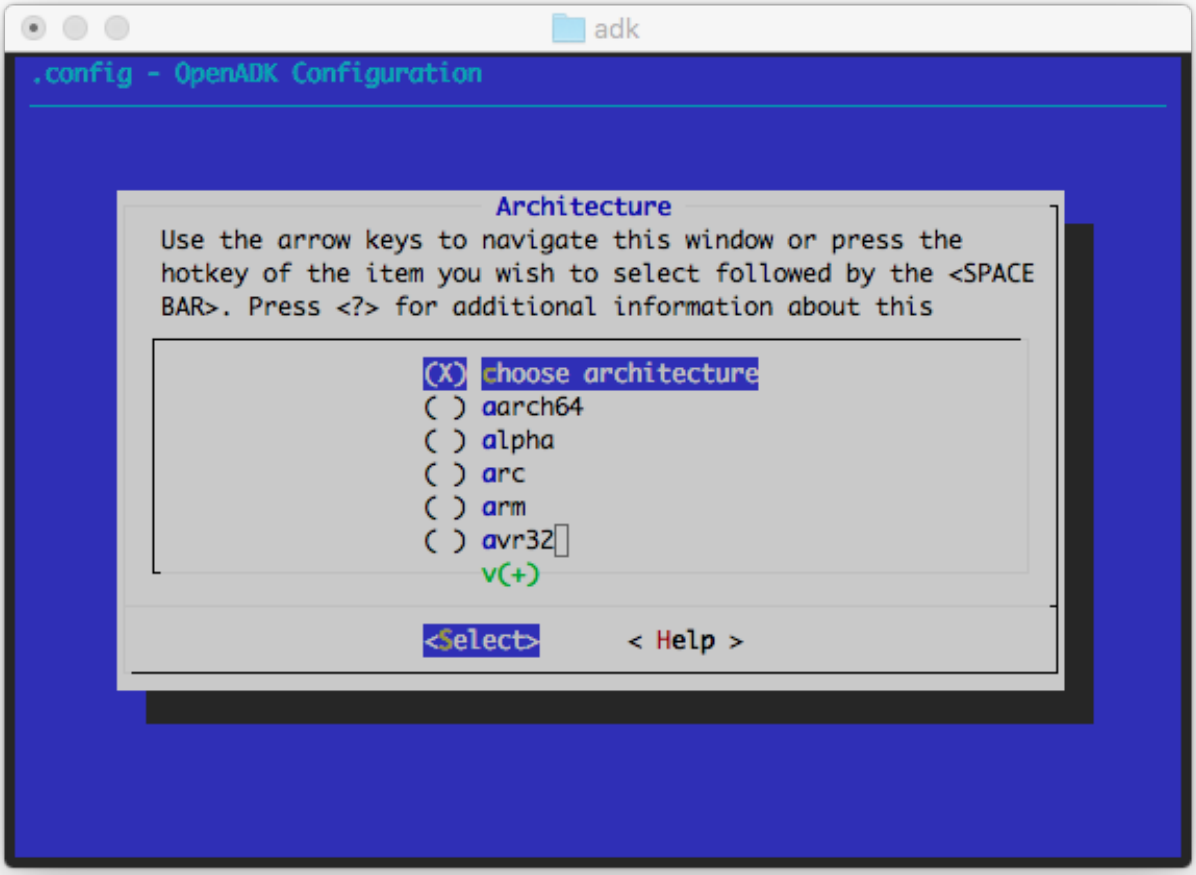
OpenADK has a nice configuration tool similar to the one you can find in the Linux kernel or in Busybox. Note that you can **and should build everything as a normal user**. There is no need to be root to configure and use OpenADK. The first step is to run the configuration assistant:

```
$ make menuconfig
```

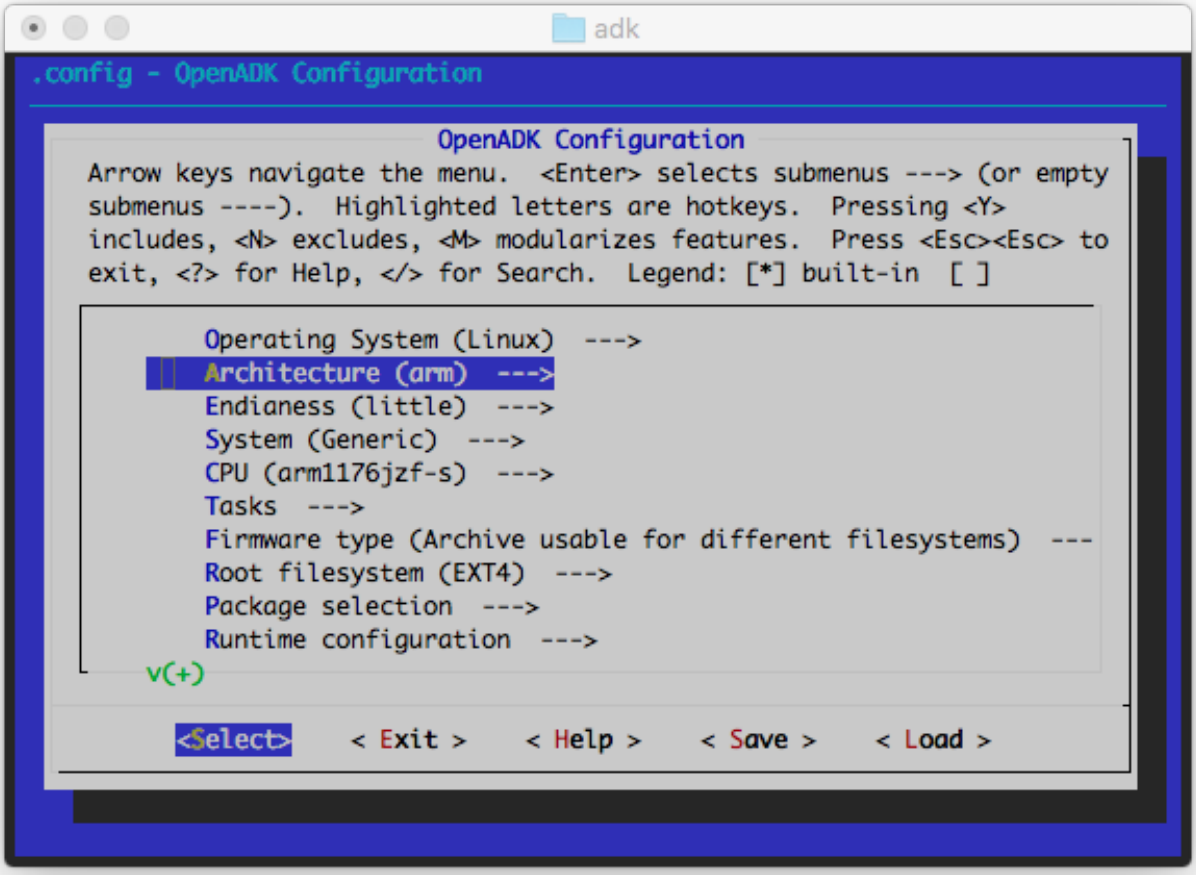
For each menu entry in the configuration tool, you can find associated help that describes the purpose of the entry.



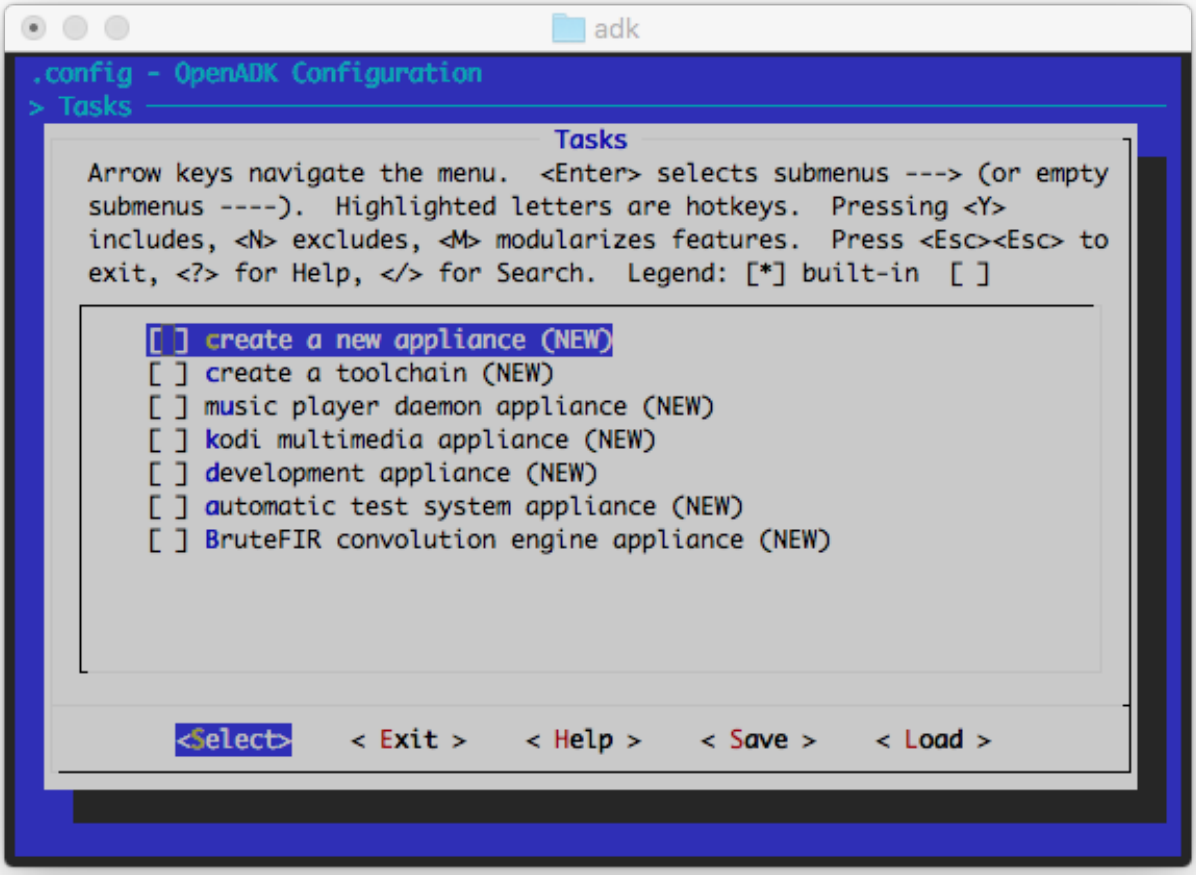
First of all you need to choose if you want to build a Linux firmware or a bare-metal toolchain. Linux is choosen as default.



After that you should select your target architecture.



Now you can select your target system, endianess, cpu and other stuff.



If you want to compile some predefined appliance tasks, you can select it in `Tasks`. You can later simply add your own tasks, which is a collection of options, packages, kernel modules or features, runtime configuration and more.

When you are ready exit and save. You can always redefine the configuration using `make menuconfig`.

Once everything is configured, the configuration tool generates a `.config` file that contains the description of your configuration. It will be used by the Makefiles to do what's needed.

Let's go:

```
$ make
```

You **should never** use `make -jN` with OpenADK: it does not support *top-level parallel make*. Instead, use the `ADK_MAKE_JOBS` option in `Global settings` to tell OpenADK to run each package compilation with `make -jN`.

The `make` command will generally perform the following steps:

- download source files
- configure, build and install required host tools
- configure, build and install the cross-compiling toolchain
- build a kernel image, if selected
- build/install selected target packages

- build a bootloader, if selected
- create a root filesystem in selected format

OpenADK output is stored in several subdirectories:

- `firmware/` where all the images and packages are stored.
- `build_<system>_<libc>_<arch>_<abi>/` where all the components except for the cross-compilation toolchain are built. The directory contains one subdirectory for each of these components.
- `target_<system>_<libc>_<arch>_<abi>/` which contains a hierarchy similar to a root filesystem hierarchy. This directory contains the installation of the cross-compilation toolchain and all the userspace packages selected for the target. However, this directory is *not* intended to be the root filesystem for the target: it contains a lot of development files, unstripped binaries and libraries that make it far too big for an embedded system. These development files are used to compile libraries and applications for the target that depend on other libraries.
- `root_<system>_<libc>_<arch>_<abi>/` which contains the complete root filesystem for the target. One exception, it doesn't have the correct permissions (e.g. setuid for the busybox binary) for some files. Therefore, this directory **should not be used on your target**. Instead, you should use one of the images or archives built in the `firmware/` directory. If you need an extracted image of the root filesystem for booting over NFS, then use the tarball image generated in `firmware/` and extract it as root. Compared to `build_*/`, `target_*/` contains only the files and libraries needed to run the selected target applications: the development files are (exception: if any dev packages are selected) not present, the binaries are stripped.
- `host_<gnu_host_name>/` contains the installation of tools compiled for the host that are needed for the proper execution of OpenADK
- `host_build_<gnu_host_name>/` contains the build directories of tools compiled for the host that are needed for the proper execution of OpenADK
- `toolchain_<system>_<libc>_<arch>_<abi>>/` contains just the cross-compilation toolchain. Can be used together with `target_<system>_<libc>_<arch>_<abi>/` for other projects. Toolchain is relocatable.
- `toolchain_build_<system>_<libc>_<arch>_<abi>/` contains the build directories for the various components of the cross-compilation toolchain.
- `pkg_<system>_<libc>_<arch>_<abi>/` contains stamp files and file lists for the various components.

The command, `make menuconfig` and `make`, are the basic ones that allow to easily and quickly generate images fitting your needs, with all the applications you enabled.

More details about the "make" command usage are given in Section 3.2.

Chapter 3

Working with OpenADK

This section explains how you can customize OpenADK to fit your needs.

3.1 Cross-compilation toolchain

A compilation toolchain is the set of tools that allows you to compile code for your system. It consists of a compiler, binary utils like assembler and linker and a C standard library.

The system installed on your development station certainly already has a compilation toolchain that you can use to compile an application that runs on your system. If you're using a PC, your compilation toolchain runs on an x86 processor and generates code for an x86 processor. Under most Linux systems, the compilation toolchain uses the GNU libc (glibc) as the C standard library. This compilation toolchain is called the "host compilation toolchain". The machine on which it is running, and on which you're working, is called the "host system" [1].

The compilation toolchain is provided by your distribution, and OpenADK has nothing to do with it (other than using it to build a cross-compilation toolchain and other tools that are run on the development host).

As said above, the compilation toolchain that comes with your system runs on and generates code for the processor in your host system. As your embedded system has a different processor, you need a cross-compilation toolchain - a compilation toolchain that runs on your *host system* but generates code for your *target system* (and target processor). For example, if your host system uses x86 and your target system uses ARM, the regular compilation toolchain on your host runs on x86 and generates code for x86, while the cross-compilation toolchain runs on x86 and generates code for ARM.

You can choose between three C libraries: uClibc-ng, glibc, musl and newlib.

There are some configuration options provided in `Toolchain settings`. You can enable or disable the building of following components and toolchain options:

- Optimization level
- Stack Smashing Protection (SSP) support
- Position Independent Executable (PIE) support
- Link Time Optimization (LTO) support
- GNU Hashstyle support
- GOLD LD support

[1] This terminology differs from what is used by GNU configure, where the host is the machine on which the application will run (which is usually the same as target)

3.2 *make* tips

This is a collection of tips that help you make the most of OpenADK.

Configuration searches: The `make menuconfig` command offer a search tool. The search tool is called by pressing `/`; The result of the search shows the help message of the matching items.

Display all commands executed by make:

```
$ make v
```

or

```
$ make ADK_VERBOSE=1 <target>
```

Display all available targets:

```
$ make help
```

Cleaning: There are different cleaning targets available. If a full clean is necessary, you normally will get a message from OpenADK. To delete all build products (including build directories, target, host and pkg trees, the firmware and the toolchain for all targets):

```
$ make cleandir
```

If you even want to clean any downloaded source and your configuration `.config`:

```
$ make distclean
```

If you only want to clean the kernel build, because you added or removed some patch, just do:

```
$ make cleankernel
```

If you just want to clean all packages and wants to rebuild the firmware, (the toolchain is not deleted) just use:

```
$ make clean
```

Resetting OpenADK for a new target: Delete the configuration and start from scratch:

```
$ rm .config*
$ make menuconfig
```

OpenADK is designed to have multiple architectures and embedded system combinations configured and build without a need to rebuild everything from scratch. There is no limit, you just need to have enough disk space.

3.3 Customization

3.3.1 Customizing the generated target filesystem

Besides changing one or another configuration through `make menuconfig`, there is a way to customize the resulting target filesystem.

Create a new directory called `extra` in the top OpenADK directory. Put there a tree of directories and files that will be copied directly over the target filesystem (`root_*`) after everything is build, but before the firmware images or archives are created.

You can also point to another directory via:

```
$ make extra=/foo/bar
```

You can start with the example configuration files from `root_*`. The `extra` directory will never be deleted by any clean target to avoid loss of customized configuration data.

3.3.2 Customizing the Busybox configuration

Busybox is very configurable, and you may want to customize it. You can just configure it via `Package selection`, `Base System`, `Busybox Configuration`. The menu based busybox configuration is mostly integrated into OpenADK. There are some options, which are not available and not supported. If you need to, you can change the defaults in `package/busybox/config` and regenerate your OpenADK configuration.

A change in the busybox configuration will rebuild the busybox package. If you choose another implementation of f.e. tar, which is provided by default from busybox, tar in busybox will be deactivated and the package will be automatically rebuilt, so that your resulting firmware images or archives will only contain a single tar program. Obviously just the one you have selected.

3.3.3 Customizing the libc configuration

Just like BusyBox Section 3.3.2 uClibc-ng offering a lot of configuration options. They allow you to select various functionalities depending on your needs and limitations. OpenADK chooses automatically the best configuration regarding resulting code size, standard conformance, portability and GNU libc compatibility.

If you still have the requirements to change the default, regenerate a new uClibc-ng config from the existing one:

```
$ tar xvf dl/uClibc-ng-x.y.z.tar.xz
$ cd uClibc-ng-x.y.z
$ cp ../target/<arch>/uclibc-ng.config .config
$ make menuconfig
```

Make all required changes. Then copy the newly created uClibc-ng configuration back and rebuild your targetsystem, including the toolchain components:

```
$ cp .config ../target/<arch>/uclibc-ng.config
$ cd .. && make cleandir && make
```

There are no customization options for GNU libc or musl available.

3.3.4 Customizing the Linux kernel configuration

The Linux kernel can be configured in the following manners by choosing the desired "Kernel configuration" option in the OpenADK configuration menu:

- using `make menuconfig` in conjunction with an OpenADK minimal configuration
- choosing a Linux kernel in-tree default configuration
- providing an extern kernel configuration file

Choosing the first option, OpenADK uses a combination of Linux miniconfig feature and user defined features to generate a valid Linux configuration for your target. Some features and drivers are not selectable via `make menuconfig`, either because your choosen target system does not have support for it or the option is not implemented, yet. OpenADK uses some kind of abstraction layer between the real full featured and complicated Linux kernel configuration and you. It is not perfect and does include a lot of manual work in `target/linux/config`, but it works in an acceptable way.

If you just want to view the Linux configuration, which is actually used for your target, you can execute following command:

```
$ make kernelconfig
```

Any changes here will get lost and will not be used to generate a kernel for your target. If you want to change the existing kernel configuration you need to follow these steps.

The basic kernel configuration used for your choosen target is concatenated from following two files: `target/linux/kernel.config` and `target/<arch>/kernel/<system>`.

So if you would like to change any basic stuff, just edit the files and recreate your firmware via:

```
$ make
```

OpenADK automatically recognizes any change and will rebuild the kernel.

The base kernel configuration for your target generated by OpenADK is normally just enough to bootup the system with support for your board, serial console, network card and boot medium. (like a hard disk, sd card or flash partition)

If you need to enable some new optional drivers or features, which are not available in `make menuconfig`, you need to dig in `target/linux/config`. There is the abstraction layer for the real kernel configuration.

The defconfig option will choose a kernel in-tree default configuration specific to your target architecture. You won't be able to do further customization.

Choosing the external configuration option, the OpenADK menu will prompt for the location of a Linux `.config` file relative to the OpenADK root directory. You will be able to alter the configuration by `make kernelconfig`. But the changes will get lost unless you save your changes by executing

```
$ make savekconfig
```

after completing the Linux kernel configuration dialog. Despite this is the most flexible way to configure the kernel, keep in mind that you are fully responsible to enable all kernel features needed to mount your filesystems and required by your applications.

3.3.5 Customizing the toolchain

There is no simple way to change anything for the toolchain. OpenADK chooses the best combination of the toolchain components to provide you with a working and recent system.

If you like to change the version of a component, add patches or like to change the configure options, you need to dig into the `toolchain` directory.

For example to change the version of gcc, you need to change `toolchain/gcc/Makefile.inc`. Be aware of the fact, that this is used for the `package/gcc/Makefile` and therefore for the gcc running on your target.

OpenADK supports running a cross-compiled toolchain on your target. You can even use OpenADK buildsystem on your target. There is a package collection called `development`, which does configure OpenADK to include all required software to use OpenADK on your target.

3.3.6 Storing the configuration

When you have a OpenADK configuration that you are satisfied with and you want to share it with others, put it under revision control or move on to a different OpenADK project.

You just need to copy your .config and extra directory to regenerate your firmware images on another system. The used config is, if not explicitely disabled, saved on the target in `/etc/adkconfig.gz`.

3.3.7 Keeping customizations outside OpenADK

The OpenADK project recommends and encourages upstreaming to the official OpenADK version the packages and board support that are written by developers. However, it is sometimes not possible or desirable because some of these packages or board support are highly specific or proprietary.

In this case, OpenADK users are offered following choice using here own git repository.

- Initialize your project

Personalize your Git environment via:

```
$ git config --global user.name "Waldemar Brodkorb"
$ git config --global user.email wbx@openadk.org
```

Get the latest version of OpenADK via anonymous git:

```
$ git clone --bare git://openadk.org/git/openadk myadk.git
```

Use git-daemon to make the repository public to your developers. After that clone your new shared project repository:

```
$ git clone git+ssh://myserver.com/git/myadk.git
$ cd myadk
```

Configure OpenADK remote git repository:

```
$ git remote add openadk git://openadk.org/git/openadk
```

- Create your firmware

Now you can either start with the latest version or use some older version:

```
$ git checkout -b stable_1_0 $sha1
```

You can find $sha1 via git log. $sha1 is the hash after the keyword “commit”.

Now build a firmware image for your target and test it. Fix bugs in the build environment or add new stuff. You can use the “extra” directory to add local unpackaged binaries and/or configuration files to overwrite packaged stuff.

Check your uncommitted changes:

```
$ git status
$ git diff --cached
$ git diff
```

Commit your git-added changes:

```
$ git commit
```

Or just commit all changes:

```
$ git commit -a
```

It is a good style to make a lot of small atomic commits.

Push your changes back to your git repository. For new local branches:

```
$ git push origin stable_1_0
```

Or in regulary usage via:

```
$ git push
```

- Working together with OpenADK

You can generate patches from all your changes against the remote master:

```
$ git format-patch -s origin
```

Send all relevant patches to OpenADK author via eMail.

Update your master with changes from OpenADK:

```
$ git checkout master
$ git pull openadk master
```

If you want you can merge all changes to your branch:

```
$ git checkout stable_1_0
$ git merge master
```

Or just cherry-pick some of the commits:

```
$ git cherry-pick $sha1
```

- Releasing

Tag your tested stable branch:

```
$ git tag -a stable_1.0
```

Push your tag to your repository:

```
$ git push origin stable_1.0
```

Checkout your tag and build your firmware:

```
$ git clone git+ssh://myserver.com/git/myadk.git mytag
$ cd mytag
$ git checkout stable_1.0
```

- Deleting unused branches

Deleting branches remotely:

```
$ git branch -r
$ git push origin :branchname
```

Deleting branches locally:

```
$ git branch
$ git branch -D branchname
```

3.4 Daily use

3.4.1 Understanding when a full rebuild is necessary

OpenADK tries to detect what part of the system should be rebuilt when the system configuration is changed through `make menuconfig`. In some cases it automatically rebuilt packages, but sometimes just a warning is printed to the terminal, that a rebuild is necessary to make the changes an effect. If strange things are happening, the autodetection might have not worked correctly, then you should consider to rebuild everything. If you are following development you should always do a full rebuild after fetching updates via `git pull`. It is not always required, but if anything fails, you are on your own. Use following to do a full rebuild without refetching distfiles:

```
$ make cleandir && make
```

3.4.2 Understanding how to rebuild packages

In OpenADK you can rebuild a single package with:

```
$ make package=<pkgname> clean package
```

It will automatically remove all files added to the staging target directory `target_*`. If you just want to restart compilation process, after making some fixes in `build_*/w-<pkgname>-<pkgversion>/`, just do:

```
$ make package=<pkgname> package
```

If you are happy with your changes to the package sources, you can automatically generate a patch, which will be saved in `package/<pkgname>/patches` and automatically applied on the next clean rebuild:

```
$ make package=<pkgname> update-patches
```

The newly created patches will be opened in $EDITOR, so you can add some comments to the top of the file, before the diff.

3.4.3 Offline builds

If you intend to do an offline build and just want to download all sources that you previously selected in the configurator then issue:

```
$ make download
```

You can now disconnect or copy the content of your `dl` directory to the build-host.

3.4.4 Environment variables

OpenADK also honors some environment variables, when they are passed to `make`.

- `ADK_APPLIANCE`, the appliance task you want to build
- `ADK_TARGET_ARCH`, the architecture of the target system
- `ADK_TARGET_SYSTEM`, the embedded target system name
- `ADK_TARGET_LIBC`, the C library for the target system
- `ADK_VERBOSE`, verbose build, when set to 1

An example that creates a configuration file for Raspberry PI with all software packages enabled, but not included in the resulting firmware image:

```
$ make ADK_APPLIANCE=new ADK_TARGET_ARCH=arm ADK_TARGET_SYSTEM=raspberry-pi  ↩
    ADK_TARGET_LIBC=musl allmodconfig
```

This is often used in the development process of a target system, to verify that all packages are compilable.

3.5 Hacking OpenADK

If OpenADK does not yet fit all your requirements, you may be interested in hacking it to add:

- new embedded targets: refer to the Adding new boards Section 7.2
- new packages: refer to the Adding new packages Section 7.3

Chapter 4

Frequently Asked Questions & Troubleshooting

4.1 Why is there no documentation on the target?

Because OpenADK mostly targets *small* or *very small* target hardware with limited resource onboard (CPU, ram, mass-storage), it does not make sense to waste space with the documentation data.

If you need documentation data on your target anyway, then OpenADK is not suitable for your purpose, and you should look for a *real distribution*.

4.2 Why is there no locale support on the target?

OpenADK tries to create a simple and small Linux system, which has no fancy features enabled. Locale support on a headless system, like a router is not useful anyway. To avoid bloat, it is a design decision to not have any locale support. Developers and users still could add any kind of user interface with internationalization features.

4.3 Why are some packages not visible in the OpenADK config menu?

If a package exists in the OpenADK tree and does not appear in the config menu, this most likely means that some of the package's dependencies are not met.

To know more about the dependencies of a package, search for the package symbol in the config menu (see Section 3.2).

Then, you may have to recursively enable several options (which correspond to the unmet dependencies) to finally be able to select the package.

If the package is not visible due to some unmet dependency to another C library, either consider to switch to another C library or fix the package so that it works with your configured library. For this you need to add your C library to PKG_LIBC_DEPENDS in `package/<pkgname>/Makefile`.

4.4 Why is there no web based configuration interface available?

OpenADK provides a basic root filesystem for your embedded device. If you need a web based configuration interface for your own appliance, just write one. There are plenty of possibilities.

Chapter 5

Running OpenADK created Linux firmware

5.1 Bootloader

The Bootloader is used to initialize the machine and load the Linux kernel. A list of popular Bootloaders can be found on http://elinux.org/Bootloader. OpenADK provides the Bootloader if necessary for a target system. You can find them in `make menuconfg` under `Packages/Bootloader`. Some Bootloaders require the Linux kernel in a special format (SREC, ELF, ..), compressed or with a special header. This will be automatically done by OpenADK in `target/<arch>/Makefile` while creating the firmware archives or images.

5.2 Linux kernel

The kernel is a program that constitutes the central core of a computer operating system. It has complete control over everything that occurs in the system. The Bootloader can provide some basic runtime configuration parameters via the kernel commandline feature.

The Linux kernel in OpenADK is intended to be very small in size and will be by default compressed with xz compression algorithm, if available for the target system. You can configure the compression algorithm used for the compression of the Linux kernel and if choosen the initramfs filesystem in `make menuconfig`. In `Linux Kernel configuration` you have the choice between dfferent kernel versions. Depending on your target devices, their might be some external git repositories available, if the support for the device is not upstream. There you can choose any needed addon drivers or any supported runtime and debugging features.

The kernel expands itself on boot, if compressed, and then initialize the hardware. The additional kernel modules are loaded later by an init script. The kernel will automatically mount the virtual filesystem /dev as devtmpfs and then will execute `/sbin/init` in userspace.

5.3 init system

The *init* program is the first userspace program started by the kernel (it carries the PID number 1), and is responsible for starting the userspace services and programs (for example: web server, graphical applications, other network servers, etc.).

In OpenADK you can choose between different init implementations. **Busybox** init is the best tested one and the default. Amongst many programs, Busybox has an implementation of a basic `init` program, which is sufficient for most embedded systems. The Busybox `init` program will read the `/etc/inittab` file at boot to know what to do. The syntax of this file can be found in http://git.busybox.net/busybox/tree/examples/inittab (note that Busybox `inittab` syntax is special: do not use a random `inittab` documentation from the Internet to learn about Busybox `inittab`). The default `inittab` in OpenADK is generated while producing the `base-files` package. The main job the default inittab does is to start the `/etc/init.d/rcS` shell script, and start one or more `getty` programs (which provides a login prompt).

Support for systemd and s6 is very experimental at the moment.

5.4 /dev management

On a Linux system, the `/dev` directory contains special files, called *device files*, that allow userspace applications to access the hardware devices managed by the Linux kernel. Without these *device files*, your userspace applications would not be able to use the hardware devices, even if they are properly recognized by the Linux kernel.

In OpenADK you can choose between different types of device managements. OpenADK defaults to **dynamic device nodes using devtmpfs and mdev**. This method relies on the *devtmpfs* virtual filesystem in the kernel, which is enabled by default for all OpenADK generated kernels, and adds the `mdev` userspace utility on top of it. `mdev` is a program part of Busybox that the kernel will call every time a device is added or removed. Thanks to the `/etc/mdev.conf` configuration file, `mdev` can be configured to for example, set specific permissions or ownership on a device file, call a script or application whenever a device appears or disappear, etc. Basically, it allows *userspace* to react on device addition and removal events. `mdev` is also important if you have devices that require a firmware, as it will be responsible for pushing the firmware contents to the kernel. `mdev` is a lightweight implementation (with fewer features) of `udev`. For more details about `mdev` and the syntax of its configuration file, see http://git.busybox.net/busybox/tree/docs/mdev.txt.

5.5 initscripts

The /etc/init.d/rcS script will execute all shell scripts in /etc/init.d in order with the parameter `autostart`. The order is identified by the `#INIT` comment in the script. All scripts are sourcing the `/etc/rc.conf` file to determine if a service should be started on boot and which flags if any are used for the service. By default all services are disabled. If the variable for a service is set to "DAEMON" and mksh is installed, the service starts asynchronously in the background. Most scripts provided by OpenADK via `package/<pkgname>/files/<pkgname>.init` are like:

```
#!/bin/sh
#PKG foo
#INIT 60

. /etc/rc.conf

case $1 in
autostop) ;;
autostart)
        test x"${foo:-NO}" = x"NO" && exit 0
        test x"$foo" = x"DAEMON" && test -x /bin/mksh && exec mksh -T- $0 start
        exec sh $0 start
        ;;
start)
        /usr/sbin/foo $foo_flags
        ;;
stop)
        kill $(pgrep -f /usr/sbin/foo )
        ;;
restart)
        sh $0 stop
        sh $0 start
        ;;
*)
        echo "usage: $0 (start|stop|restart)"
        exit 1
esac
exit $?
```

5.6 cfgfs - configuration file system

The cfgfs application for the OpenADK system uses a special small partition on the block device of your embedded system (f.e. flash, sd card, compact flash or hard disk). Only changes made to /etc on your embedded system are saved in a compressed form

(using LZO1 compression algorithm) in this partition. There is no Linux filesystem on this partition. The embedded system initialization process will setup /etc correctly on boot up, when cfgfs application is found. After making any changes to /etc, which should survive a reboot of the embedded system must be written to the cfgfs partition via "cfgfs commit". Trying to reboot, shutdown or halt an embedded system with unsaved changes will generate an error, which can be circumvented. Updates to /etc via a package manager (f.e. ipkg) will be reported.

```
cfgfs
Configuration Filesystem Utility (cfgfs)
Syntax:
        /sbin/cfgfs commit [-f]
        /sbin/cfgfs erase
        /sbin/cfgfs setup [-N]
        /sbin/cfgfs status [-rq]
        /sbin/cfgfs { dump | restore } [<filename>]
```

5.7 network configuration

On bootup `/etc/network/interfaces` is used to find out which network configuration should be used. The default is to use DHCP (via busybox `udhcpc`) on the first found ethernet device to configure the network. See network configuration for detailed syntax of `/etc/network/interfaces`. It is similar to Debian network configuration and uses `ifupdown` from `busybox`.

See Appendix Section 10.1

5.8 getting a shell on the system

There are two methods available to get a shell on your embedded system created with OpenADK. You can either login locally via serial console or graphical console or you can login remotely via secure shell.

In both cases the default user is `root` and the default password is `linux123`. **You should always change the default password!!** You can do this either via `passwd` on the system or you can preconfigure a password via `make menuconfig` under `Runtime configuration`.

The default shell used in OpenADK is `mksh` from http://www.mirbsd.org/mksh/. You can change the shell in `make menuconfig` under `Runtime configuration`. Be aware of the fact that the bootup process might use some `mksh` features to speedup the system start. When you change the shell for system `/bin/sh` the slower startup is used as a fallback.

Chapter 6

Going further in OpenADK's innards

6.1 How OpenADK works

As mentioned above, OpenADK is basically a set of Makefiles that download, configure, and compile software with the correct options. It also includes patches for various software packages and the Linux kernel.

There is basically one Makefile per software package. Makefiles are split into many different parts.

- The `toolchain/` directory contains the Makefiles and associated files for all software related to the cross-compilation toolchain: `binutils`, `gcc`, `gdb`, `kernel-headers` and `libc`.
- The `target/` directory contains the definitions for all the processor architectures that are supported by OpenADK. `target/linux` contains the meta-data for the Linux kernel configuration abstraction layer and the kernel patches.
- The `package/` directory contains the Makefiles and associated files for all user-space tools and libraries that OpenADK can compile and add to the target root filesystem or to the host directory. There is one sub-directory per package.
- The `mk/` directory contains some globally used Makefiles with the suffix `.mk`, these are used in all other Makefiles via include.
- The `adk/` directory contains the Makefiles and associated files for software related to the generation of the host tools needed for `make menuconfig` system.
- The `scripts/` directory contains shell scripts for the creation of meta-data in OpenADK, install scripts and image creation scripts.

The configuration process is separated in following steps:

- Makefile is just a wrapper which calls the prerequisite shell script.
- The prerequisite shell script `scripts/prereq.sh` checks if the host system have all required software installed and tries to build GNU make and bash if it is missing. It creates the `prereq.mk` Makefile.
- Compile and run the OpenADK tools to generate the meta-data for the menu based configuration and creates the `package/Depends.mk` Makefile to handle the dependencies.
- Starts the menu based configuration system via `make menuconfig`.

The following steps are performed, once the configuration is done (mainly implemented in `mk/build.mk`):

- Create all the output directories: `host_<gnu_host_name>`, `target_<arch>_<libc>`, `build_<arch>_<libc>`, `pkg_<arch>_<libc>`, etc.
- Generate the host tools required for different tasks (encrypting passwords, compressing data, extracting archives, creating images, ..)

- Generate the cross-compilation toolchain (binutils, gcc, libc, gdb)
- Compile the Linux kernel
- Compile all the userspace packages, the boot loader and external kernel modules
- Generate the firmware images or archives
- Output a target specific help text

6.2 Advanced usage

6.2.1 Using the generated toolchain outside OpenADK

You may want to compile, for your target, your own programs or other software that are not packaged in OpenADK. In order to do this you can use the toolchain that was generated by OpenADK.

The toolchain generated by OpenADK is located by default in `toolchain_<gnu_host_name>/`. The simplest way to use it is to add `toolchain_<gnu_host_name>/usr/bin/` to your PATH environment variable and then to use `<arch>-<vendor>-linux-<libcsuffix>-gcc`, `<arch>-<vendor>-linux-<libcsuffix>-objdump`, etc.

It is possible to relocate the toolchain, you just need to put `target_<arch>_<libc>_<libcsuffix>` into the same directory as `toolchain_<gnu_host_name>/`.

6.2.2 Using `ccache` in OpenADK

ccache is a compiler cache. It stores the object files resulting from each compilation process, and is able to skip future compilation of the same source file (with same compiler and same arguments) by using the pre-existing object files. When doing almost identical builds from scratch a number of times, it can nicely speed up the build process.

`ccache` support is integrated in OpenADK. You just have to enable `Use ccache to speedup recompilation` in `Globale settings`. This will automatically build `ccache` and use it for every target compilation.

The cache is located in `$HOME/.ccache`. It is stored outside of OpenADK directory so that it can be shared by separate OpenADK builds. If you want to get rid of the cache, simply remove this directory.

6.2.3 Location of downloaded packages

The various tarballs that are downloaded by OpenADK are all stored in `ADK_DL_DIR`, which by default is the `dl` directory. If you want to keep a complete version of OpenaDK which is known to be working with the associated tarballs, you can make a copy of this directory. This will allow you to regenerate the toolchain and the target filesystem with exactly the same versions.

If you maintain several OpenADK trees, it might be better to have a shared download location. This can be achieved by pointing the `DL_DIR` environment variable to a directory. If this is set, then the value of `ADK_DL_DIR` in the OpenADK configuration is overridden. The following line should be added to `<~/.bashrc>`.

```
$ export DL_DIR=<shared download location>
```

The download location can also be set in the `.config` file, with the `ADK_DL_DIR` option. Unlike most options in the .config file, this value is overridden by the `DL_DIR` environment variable.

6.2.4 Package-specific *make* targets

Running `make package=<package> package` builds and installs that particular package. Be aware of the fact, that no build dependencies are resolved using this method!

For packages relying on the OpenADK infrastructure, there are numerous special make targets that can be called independently like this:

```
$ make package=<package> <target>
```

The package build targets are (in the order they are executed):

command/target	Description
fetch	Fetch the source
extract	Put the source in the package build directory
patch	Apply the patches, if any
configure	Run the configure commands, if any
build	Run the compilation commands
fake	Run the installation of the package into a fake directory
package	Create a package or tar archive of the package files

Additionally, there are some other useful make targets:

command/target	Description
clean	Remove the whole package build directory
hostclean	Remove the whole hostpackage build directory
hostpackage	Build and install the host binaries and libraries

6.2.5 Using OpenADK during development

The normal operation of OpenADK is to download a tarball, extract it, configure, compile and install the software component found inside this tarball. The source code is extracted in build_<system>_<arch>_<libc>/w-<package>-<version>, which is a temporary directory: whenever make clean or one of the other clean targets are used, this directory is entirely removed, and recreated at the next make invocation.

This behavior is well-suited when OpenADK is used mainly as an integration tool, to build and integrate all the components of an embedded Linux system. However, if one uses OpenADK during the development of certain components of the system, this behavior is not very convenient: one would instead like to make a small change to the source code of one package, and be able to quickly rebuild the system with OpenADK.

Following workflow might help to integrate your own changes, while developing a new package or board support.

Make changes directly in build_<system>_<arch>_<libc>/w-<package>-<version> and recompile the package with:

```
$ make package=<package> package
```

When you are happy with the change, generate a patch:

```
$ make package=<package> update-patches
```

For the linux kernel just change the code in +build_<system>_<arch>_<libc>/linux, remove the .config and call make again:

```
$ rm build_<system>_<arch>_<libc>/linux/.config
$ make
```

There is no update-patches target for the kernel, you need to extract the kernel source from your download dir, make a copy of the source tree, add your changes and create a patch manually:

```
$ tar xvf dl/linux-x.y.z.tar.xz
$ cp -a linux-x.y.z linux-x.y.z.orig
$ diff -Nur linux-x.y.z.orig linux-x.y.z > target/linux/patches/x.y.z/mykernel.patch
$ make cleankernel
$ make
```

The same method can be used for toolchain components and *must* be used for busybox, because it contains patches, which are not generated via make update-patches.

Chapter 7

Developer Guidelines

7.1 Coding style

Overall, these coding style rules are here to help you to add new files in OpenADK or refactor existing ones.

7.1.1 `Config.in`

`Config.in` files contain entries for almost anything configurable in OpenADK. Mostly all Config.in files for packages are autogenerated and should not be manually edited. The following rules apply for the top level Config.in, for the files in target/config and target/linux/config.

An entry has the following pattern:

```
config ADK_TARGET_FOO
        bool "foo"
        select BR2_PACKAGE_LIBBAR
        depends on ADK_PACKAGE_LIBBAZ
        default n
        help
          This is a comment that explains what foo is.

          http://foo.org/foo/
```

- The `bool`, `depends on`, `default`, `select` and `help` lines are indented with one tab.
- The help text itself should be indented with one tab and two spaces.

The `Config.in` files are the input for the configuration tool used in OpenADK, which is an enhanced version of *Kconfig*. For further details about the *Kconfig* language, refer to http://kernel.org/doc/Documentation/kbuild/kconfig-language.txt.

7.1.2 `Makefile`

- Header: The file starts with a license header.

```
# This file is part of the OpenADK project. OpenADK is copyrighted
# material, please see the LICENCE file in the top-level directory.
```

- Assignment: use `:=` or `+=` followed by two tabs:

```
PKG_VERSION:=           1.0
PKG_BUILDDEP+=          libfoo
```

- Indentation: use tab only:

```
libfoo-install:
	$(CP) $(WRKINST)/usr/lib/libfoo*.so* \
		$(IDIR_LIBFOO)/usr/lib
```

- Optional dependency:
  - Prefer multi-line syntax.

```
ifeq ($(ADK_PACKAGE_LIBFOO_WITH_PYTHON),y)
CONFIGURE_ARGS+=	--with-python-support
else
CONFIGURE_ARGS+=	--without-python-support
endif
```

7.1.3 Documentation

The documentation uses the asciidoc format.

For further details about the asciidoc syntax, refer to http://www.methods.co.nz/asciidoc/userguide.html.

7.2 Adding new embedded boards to OpenADK

This section covers how support for new embedded boards can be integrated into OpenADK.

First step is to create a board description file in target/<arch>/systems with the short name of your embedded board.

For example you would create following file for Raspberry PI 2 support: target/arm/systems/raspberry-pi2

```
config ADK_TARGET_SYSTEM_RASPBERRY_PI2
	bool "Raspberry PI 2"
	depends on ADK_TARGET_LITTLE_ENDIAN
	select ADK_TARGET_CPU_ARM_CORTEX_A7
	select ADK_TARGET_CPU_WITH_NEON
	select ADK_TARGET_BOARD_BCM28XX
	select ADK_TARGET_WITH_VGA
	select ADK_TARGET_WITH_SERIAL
	select ADK_TARGET_WITH_CPU_FREQ
	select ADK_TARGET_WITH_USB
	select ADK_TARGET_WITH_INPUT
	select ADK_TARGET_WITH_SD
	select ADK_TARGET_WITH_I2C
	select ADK_TARGET_WITH_SPI
	select ADK_TARGET_WITH_SMP
	select ADK_PACKAGE_BCM28XX_BOOTLOADER
	select ADK_TARGET_WITH_ROOT_RW
	select ADK_TARGET_KERNEL_ZIMAGE
	help
	  Raspberry PI 2
```

You need to select as a minimum a CPU type and Kernel format. If a bootloader is required you also need to select it. (ADK_PACKAGE_BCM28XX_BOOTLOADER) If the bootloader does not exist as a package in OpenADK, you need to port it first.

The hardware capabilities are optional. (f.e. ADK_TARGET_WITH_SD), but required when you configure the driver configuration later.

For architectures with a choice for endianess you should depends on either ADK_TARGET_LITTLE_ENDIAN or ADK_TARGET_BIG

If the CPU type like in this example ADK_TARGET_CPU_ARM_CORTEX_A7 is not yet available you need to add it to target/-config/Config.in.cpu. For optimized code generation you should also add ADK_TARGET_GCC_CPU or ADK_TARGET_GCC_ARCH symbol for your CPU type. Furthermore you need to decide if your CPU has a MMU, FPU and NPTL support in the C library.

After the creation of the file you can go into the menu based system and select your embedded board.

The second step is to create a Kernel configuration file fragment, which contains only the basic support for your board to get serial console access.

For example the snippet for Raspberry PI 2, the file name must match the embedded board name: target/arm/kernel/raspberry-pi2

```
CONFIG_ARM=y
CONFIG_ARCH_BCM2709=y
CONFIG_BCM2709_DT=y
CONFIG_PHYS_OFFSET=0
CONFIG_HAVE_ARM_ARCH_TIMER=y
CONFIG_FIQ=y
CONFIG_ATAGS=y
CONFIG_KUSER_HELPERS=y
CONFIG_ARM_ERRATA_643719=y
CONFIG_BCM2708_NOL2CACHE=y
CONFIG_RASPBERRYPI_FIRMWARE=y
CONFIG_BRCM_CHAR_DRIVERS=y
CONFIG_BCM2708_VCHIQ=y
CONFIG_BCM2708_VCMEM=y
CONFIG_MAILBOX=y
CONFIG_BCM2835_MBOX=y
CONFIG_OF=y
CONFIG_OF_OVERLAY=y
CONFIG_CMDLINE_FROM_BOOTLOADER=y
```

If the mainstream kernel from kernel.org does not contain support for your board you need to get a working kernel tree and create a patch. For example for Raspberry PI 2 we basically use following method to create a patch:

```
git clone https://github.com/raspberrypi/linux.git linux-rpi
wget https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.18.9.tar.xz
tar xvf linux-3.18.9.tar.xz
find linux-3.18.9 linux-rpi -type l -delete
rm -rf linux-rpi/.git
diff -Nur linux-3.18.9 linux-rpi > target/arm/bcm28xx/patches/3.18.9/0001-bcm28xx-github. ↩
    patch
```

Normally you use target/<arch>/<target system>/patches/<kernelversion>/0001-<target-system>.patch. In case of Raspberry PI 2 we have a single patch for Raspberry PI and Raspberry PI 2 and use the extra board name bcm28xx to describe the family of devices.

After that you can build the toolchain, kernel and base system and write the resulting firmware from firmware/<target system>/ to your device or boot via netboot and NFS.

If you have some special notes for your embedded board, please add some advise to target/<arch>/Makefile. You can add information for the different rootfilesystem types.

If your system boots up fine to a shell, you can add the driver configuration. For example if you add SD card driver support to Raspberry PI 2 you would add following to target/linux/config/Config.in.block

```
config ADK_KERNEL_MMC_BCM2835
        bool "SD card support for BCM2835 boards"
        select ADK_KERNEL_SCSI
        select ADK_KERNEL_MMC
        select ADK_KERNEL_MMC_BLOCK
        select ADK_KERNEL_BLK_DEV
        select ADK_KERNEL_BLK_DEV_SD
        select ADK_KERNEL_MMC_SDHCI
        select ADK_KERNEL_MMC_SDHCI_PLTFM
```

```
        select ADK_KERNEL_MMC_BCM2835_DMA
        depends on ADK_TARGET_BOARD_BCM28XX
        default y if ADK_TARGET_BOARD_BCM28XX
        default n
```

We use the symbol prefix ADK_KERNEL instead of CONFIG. Otherwise the symbols are matching the kernel symbol names.

Get again into the menu based system, enable the driver you added and recompile. If your driver is available as a kernel module use tristate.

7.3 Adding new packages to OpenADK

This section covers how new packages (userspace libraries or applications) can be integrated into OpenADK. It also shows how existing packages are integrated, which is needed for fixing issues or tuning their configuration.

7.3.1 New package

First of all, create a directory and Makefile under the `package` directory for your software, for example `libfoo`:

```
$ make newpackage PKG=libfoo VER=0.1
```

This will create a sample Makefile for you, with a lot of comments and hints. It describes how the package should be downloaded, configured, built, installed, etc.

Depending on the package type, the `Makefile` must be written in a different way, using two different infrastructures:

- manual package configuration
- automatic package configuration using autotools

Dependencies on target and toolchain options

Some packages depend on certain options of the toolchain: mainly the choice of C library and C++ support. Some packages can only be built on certain target architectures or for specific target systems.

These dependencies have to be expressed in the Makefile. The given values are space separated and can be negated with ! as a prefix.

- Target architecture
  - variable used PKG_ARCH_DEPENDS
  - allowed values are: arm, mips, .. see target/arch.lst
- Target system
  - variable used PKG_SYSTEM_DEPENDS
  - for allowed values see the output of: find target/*/systems -type f
- Target C library
  - variable used PKG_LIBC_DEPENDS
  - allowed values are: uclibc-ng glibc musl
- Host system
  - variable used PKG_HOST_DEPENDS
  - allowed values are: linux darwin cygwin freebsd netbsd openbsd

- Special support needed (Toolchain with Threads, Realtime or C++ enabled)
  - variable used PKG_NEEDS
  - allowed values are: threads rt c++

Further formatting details: see the writing rules Section 7.1.2.

7.3.2 Infrastructure for packages with specific build systems

By *packages with specific build systems* we mean all the packages whose build system is not the standard one, speak *autotools*. This typically includes packages whose build system is based on hand-written Makefiles or shell scripts.

```
01: # This file is part of the OpenADK project. OpenADK is copyrighted
02: # material, please see the LICENCE file in the top-level directory.
03:
04: include $(ADK_TOPDIR)/rules.mk
05:
06: PKG_NAME:=          libfoo
07: PKG_VERSION:=       1.0
08: PKG_RELEASE:=       1
09: PKG_HASH:=          62333167b79afb0b25a843513288c67b59547acf653e8fbe62ee64e71ebd1587
10: PKG_DESCR:=         foo library
11: PKG_SECTION:=       libs
12: PKG_BUILDDEP:=      libressl
13: PKG_DEPENDS:=       libressl
14: PKG_URL:=           http://www.libfoo.org/
15: PKG_SITES:=         http://download.libfoo.org/
16:
17: include $(ADK_TOPDIR)/mk/package.mk
18:
19: $(eval $(call PKG_template,LIBFOO,libfoo,${PKG_VERSION}-${PKG_RELEASE},${PKG_DEPENDS},$ ↩
    {PKG_DESCR},${PKG_SECTION}))
20:
21: CONFIG_STYLE:=             manual
22: BUILD_STYLE:=              manual
23: INSTALL_STYLE:=            manual
24:
25: do-configure:
26:     ${CP} ./files/config ${WRKBUILD}/.config
27:
28: do-build:
29:     ${MAKE} -C ${WRKBUILD} all
30:
31: do-install:
32:     ${INSTALL_DIR} ${IDIR_LIBFOO}/usr/lib
33:     ${CP} ${WRKBUILD}/libfoo.so* ${IDIR_LIBFOO}/usr/lib
34:
35: include ${ADK_TOPDIR}/mk/pkg-bottom.mk
```

The Makefile begins with line 4 with the inclusion of the top level rules.mk file. After that the Makefile starts on line 6 to 15 with metadata information: the name of the package (`PKG_NAME`), the version of the package (`PKG_VERSION`), the release number of the package (`PKG_RELEASE`), which is used in OpenADK to mark any package updates, the sha256 hash of the source archive (`PKG_HASH`), the short one line description for the package (`PKG_DESCR`), the package section for the menu configuration system (`PKG_SECTION`), the package buildtime dependencies (`PKG_BUILDDEP`), the package runtime dependencies (`PKG_DEPENDS`), the package homepage (`PKG_URL`) and finally the internet locations at which the tarball can be downloaded from (`PKG_SITES`). overwrite the default via the `DISTFILES` variable. You can add more then one archive name in `DISTFILES` via space separated. If you have no source archive at all, just use the boolean variable `NO_DISTFILES` and set it to 1.

On line 17 the `mk/package.mk` file is included, which contains the PKG_template function, which is used in line 19.

On line 21 to 23 we define that the configuration step, the building and install steps are manually provided.

On line 25-26 we implement a manual configuration step of the libfoo package by copying a manually created config file into the build directory.

On line 28-29 we start the compilation process via make.

On line 31-33 we install the shared library into the package installation directory, which is used to create the resulting binary package or tar archive for the target.

On line 35 we include `mk/pkg-bottom.mk`, which includes common functions used by the package fetching and building process.

7.3.3 Infrastructure for autotools-based packages

First, let's see how to write a `Makefile` file for an autotools-based package, with an example:

```
01: # This file is part of the OpenADK project. OpenADK is copyrighted
02: # material, please see the LICENCE file in the top-level directory.
03:
04: include ${ADK_TOPDIR}/rules.mk
05:
06: PKG_NAME:=          libfoo
07: PKG_VERSION:=       1.0
08: PKG_RELEASE:=       1
09: PKG_HASH:=          62333167b79afb0b25a843513288c67b59547acf653e8fbe62ee64e71ebd1587
10: PKG_DESCR:=         foo library
11: PKG_SECTION:=       libs
12: PKG_BUILDDEP:=      curl
13: PKG_DEPENDS:=       libcurl
14: PKG_URL:=           http://www.libfoo.org/
15: PKG_SITES:=         http://downloads.libfoo.org/
16:
17: include ${ADK_TOPDIR}/mk/package.mk
18:
19: $(eval $(call PKG_template,LIBFOO,libfoo,${PKG_VERSION}-${PKG_RELEASE},${PKG_DEPENDS},$ ↩
    {PKG_DESCR},${PKG_SECTION}))
20:
21: libfoo-install:
22:     ${INSTALL_DIR} ${IDIR_LIBFOO}/usr/lib
23:     ${CP} ${WRKINST}/usr/lib/libfoo.so* ${IDIR_LIBFOO}/usr/lib
24:
25: include ${ADK_TOPDIR}/mk/pkg-bottom.mk
```

The Makefile begins with line 4 with the inclusion of the top level rules.mk file. After that the Makefile starts on line 6 to 15 with metadata information: the name of the package (`PKG_NAME`), the version of the package (`PKG_VERSION`), the release number of the package (`PKG_RELEASE`), which is used in OpenADK to mark any package updates, the sha256 hash of the source archive (`PKG_HASH`), the short one line description for the package (`PKG_DESCR`), the package section for the menu configuration system (`PKG_SECTION`), the package buildtime dependencies (`PKG_BUILDDEP`), the package runtime dependencies (`PKG_DEPENDS`), the package homepage (`PKG_URL`) and finally the internet locations at which the tarball can be downloaded from (`PKG_SITES`). overwrite the default via the `DISTFILES` variable. You can add more then one archive name in `DISTFILES` via space separated. If you have no source archive at all, just use the boolean variable `NO_DISTFILES` and set it to 1.

On line 17 the `mk/package.mk` file is included, which contains the PKG_template function, which is used in line 19.

On line 21-23 we install the shared library into the package installation directory, which is used to create the resulting binary package or tar archive for the target.

On line 25 we include `mk/pkg-bottom.mk`, which includes common functions used by the package fetching and building process.

With the autotools infrastructure, all the steps required to build and install the packages are already defined, and they generally work well for most autotools-based packages. However, when required, it is still possible to customize what is done in any particular step. By adding a post-operation hook (after extract, patch, configure, build or install). See Section 7.3.5 for details.

7.3.4 Infrastructure for host packages

First, let's see how to write a `Makefile` for an host only package, required by another target package to build, with an example:

```
01: # This file is part of the OpenADK project. OpenADK is copyrighted
02: # material, please see the LICENCE file in the top-level directory.
03:
04: include $(ADK_TOPDIR)/rules.mk
05:
06: PKG_NAME:=              hostfoo
07: PKG_VERSION:=           1.0
08: PKG_RELEASE:=           1
09: PKG_HASH:=              62333167 ↩
    b79afb0b25a843513288c67b59547acf653e8fbe62ee64e71ebd1587
10: PKG_DESCR:=             hostfoo utility
11: PKG_SECTION:=           misc
12: PKG_URL:=               http://www.foo.org/
13: PKG_SITES:=             http://download.foo.org/
14:
15: PKG_CFLINE_HOSTFOO:=    depends on ADK_HOST_ONLY
16:
17: include $(ADK_TOPDIR)/mk/host.mk
18: include $(ADK_TOPDIR)/mk/package.mk
19:
20: $(eval $(call HOST_template,HOSTFOO,hostfoo,$(PKG_VERSION)-${PKG_RELEASE}))
21:
22: HOST_STYLE:=            auto
23:
24: include ${ADK_TOPDIR}/mk/host-bottom.mk
25: include ${ADK_TOPDIR}/mk/pkg-bottom.mk
```

The differences to a target package is the inclusion of `mk/host.mk` in line 17 and `mk/host-bottom.mk` in line 24. Furthermore the HOST_template is called instead of the PKG_template. The last difference is the usage of `PKG_CFLINE_HOSTFOO` to mark the package as host only package.

Following mix between host and target package is possible, too:

```
01: # This file is part of the OpenADK project. OpenADK is copyrighted
02: # material, please see the LICENCE file in the top-level directory.
03:
04: include ${ADK_TOPDIR}/rules.mk
05:
06: PKG_NAME:=          foo
07: PKG_VERSION:=       1.0
08: PKG_RELEASE:=       1
09: PKG_HASH:=          62333167b79afb0b25a843513288c67b59547acf653e8fbe62ee64e71ebd1587
10: PKG_DESCR:=         foo tool
11: PKG_SECTION:=       lang
12: PKG_BUILDDEP:=      foo-host
13: PKG_URL:=           http://www.foo.org/
14: PKG_SITES:=         http://download.foo.org/
15:
16: include ${ADK_TOPDIR}/mk/host.mk
17: include ${ADK_TOPDIR}/mk/package.mk
18:
19: $(eval $(call HOST_template,FOO,foo,${PKG_VERSION}-${PKG_RELEASE}))
20: $(eval $(call PKG_template,FOO,foo,${PKG_VERSION}-${PKG_RELEASE},${PKG_DEPENDS},${ ↩
    PKG_DESCR},${PKG_SECTION}))
21:
22: HOST_STYLE:=        auto
23:
24: foo-install:
```

```
25:     ${INSTALL_DIR} ${IDIR_FOO}/usr/bin
26:     ${INSTALL_BIN} ${WRKINST}/usr/bin/foo ${IDIR_FOO}/usr/bin
27:
28: include ${ADK_TOPDIR}/mk/host-bottom.mk
29: include ${ADK_TOPDIR}/mk/pkg-bottom.mk
```

If you need to rebuild a mixed package, you can do:

```
$ make package=<package> hostclean hostpackage clean package
```

If your host package have some dependencies, use following:

```
HOST_BUILDDEP:=libbaz-host bar-host
```

7.3.5 Hooks available in the various build steps

The infrastructure allow packages to specify hooks. These define further actions to perform after existing steps. Most hooks aren't really useful for manual packages, since the `Makefile` already has full control over the actions performed in each step of the package construction.

The following hook targets are available:

- `post-extract`
- `post-patch`
- `pre-configure`
- `post-configure`
- `pre-build`
- `post-build`
- `pre-install`
- `post-install`

For example, to make some scripts executable after extraction, add following to your `Makefile`:

```
post-extract:
        chmod a+x $(WRKBUILD)/build/make/*.sh
        chmod a+x $(WRKBUILD)/build/make/*.pl
```

7.3.6 Conclusion

As you can see, adding a software package to OpenADK is simply a matter of writing a Makefile using an existing template and modifying it according to the compilation process required by the package.

If you package software that might be useful for other people, don't forget to send a patch to the OpenADK developer (see Section 9.1)!

7.3.7 package Reference

The list of variables that can be set in a Makefile to give metadata information is:

- PKG_NAME, mandatory, must contain the name of the package.
- PKG_VERSION, mandatory, must contain the version of the package.
- PKG_RELEASE, mandatory, must contain the OpenADK specific release of the package.
- PKG_HASH, mandatory, must contain the SHA256 hash of the package, will be used to check if a download of a package is complete.
- PKG_SECTION, mandatory, must contain the OpenADK specific section, see package/section.lst.
- PKG_RELEASE, mandatory, must contain an one line summary of the package description.
- PKG_URL, optional, may contain the url to the homepage of the package
- PKG_SITES, mandatory, must contain the download url for the package, multiple entries with space separated, are allowed. Only HTTP/HTTPS or FTP URLS are allowed. A backup site (http://www.openadk.org/distfiles) is always used, if the package site is not available. There is no direct support for cvs/svn/git/hg/bzr repositories, because building OpenADK behind a HTTP proxy should be working without any configuration hassle. There are also some predefined mirror sites in mk/mirrors.mk, which can be used.
- DISTFILES optional, may contain the name of the tarball of the package. If DISTFILES is not specified, it defaults to PKG_NAME-PKG_VERSION.tar.xz.
- NO_DISTFILES optional, may be set to 1, to disable fetching of any archives. Provide the source code for the package in package/<pkgname>/src, which will be automatically copied to the WRKBUILD/WRKSRC directory.
- PKG_BUILDDEP optional, lists the build time dependencies (in terms of package directory name, see package/) that are required for the current target package to compile. These dependencies are guaranteed to be compiled and installed before the configuration of the current package starts.
- PKG_DEPENDS optional, lists the runtime dependencies that are required to run the software package on the target. It contains a list of package names, which might be different to the package directory name. See what is used in PKG_template, to find out the package name used here.
- PKG_KDEPENDS optional, lists the kernel module dependencies that are required to run the software package on the target. It contains a list of kernel module names in lower case as used in target/linux/config. (use minus instead of underscores)
- PKG_NEEDS optional, lists the features that are required to build or run the software package on the target. It contains a list of keywords. Supported are threads mmu intl and c++
- PKG_NOPARALLEL optional, may be set to 1, to disable parallel building of the package via make -jn, n=4 is default, but can be changed in Global Settings in the menu based configuration.
- PKG_OPTS optional, may be set to following values: dev create a development package automatically, containing header files and .pc files. Only useful for library packages, when you want to compile on the target. devonly only creates a development package with header files, normally not needed on the target. noscripts do not automatically install *-config and other build related scripts into STAGING_TARGET_DIR/scripts, required for automake/autoconf package noremove do not automatically remove package files from STAGING_TARGET_DIR

The recommended way to define these variables is to use the following syntax:

```
PKG_VERSION:=           2.11
```

Or for lines longer than 80 characters use:

```
PKG_DEPENDS:=           foo bar baz
PKG_DEPENDS+=           muh maeh
```

The variables that define what should be performed at the different steps of the configure, build and install process.

- CONFIG_STYLE manual, auto, minimal, basic, perl or cmake
- CONFIGURE_ARGS add --enable-foo/--disable-foo to configure
- CONFIGURE_ENV add additional environment variables to configure step
- HOST_STYLE either manual or auto
- HOST_CONFIGURE_ARGS add --enable-foo/--disable-foo to host configure
- HOST_CONFIGURE_ENV add additional environment variables to the host configure step
- AUTOTOOL_STYLE either autoreconf, autoconf or bootstrap
- BUILD_STYLE either manual or auto
- MAKE_ENV add additional variables to build step
- MAKE_FLAGS add additinal make flags to build step
- FAKE_FLAGS add additional make flags to fake install step
- XAKE_FLAGS add additional make flags to build and fake install step
- INSTALL_STYLE either manual or auto
- CONFIGURE_PROG overwrite default configure programm
- MAKE_FILE overwrite default Makefile
- ALL_TARGET overwrite default build target
- INSTALL_TARGET overwrite default install target

The variables to add or overwrite preprocessor, compiler and linker flags:

- TARGET_CPPFLAGS flags for the preprocessor
- TARGET_CFLAGS flags for the compiler
- TARGET_LDFLAGS flags for the linker
- TARGET_CXXFLAGS flags for the C++ compiler
- HOST_CPPFLAGS flags used for host preprocessing
- HOST_CFLAGS flags used for host compiling
- HOST_LDFLAGS flags used for host linking
- HOST_CXXFLAGS flags for the C++ host compiler

7.4 Patching a package

While integrating a new package or updating an existing one, it may be necessary to patch the source of the software to get it cross-built within OpenADK. OpenADK offers an infrastructure to automatically handle this during the builds. Patches are provided within OpenADK, in the package directory; these typically aim to fix cross-compilation, libc support, portability issues or other things.

Normally the patches are autogenerated via:

```
$ make package=<package> update-patches
```

Otherwise they are manually generated via:

```
$ diff -Nur <pkgname>-<pkgversion>.orig <pkgname>-<pkgversion> > package/<pkgname>/patches ↩
    /xxx-description.patch
```

The string `xxx` should be substituted by a number starting with 001. The patches will be applied in numeric order. You should either use the automatic patch generation or the manual patch creation for a package. Mixed usage is not supported.

7.4.1 Format and licensing of the package patches

Patches are released under the same license as the software that is modified.

A message explaining what the patch does, and why it is needed, should be added in the header commentary of the patch. At the end, the patch should look like:

```
add C++ support test

--- configure.ac.orig
+++ configure.ac
@@ -40,2 +40,12 @@

AC_PROG_MAKE_SET
+
+AC_CACHE_CHECK([whether the C++ compiler works],
+               [rw_cv_prog_cxx_works],
+               [AC_LANG_PUSH([C++])
+                AC_LINK_IFELSE([AC_LANG_PROGRAM([], [])],
+                               [rw_cv_prog_cxx_works=yes],
+                               [rw_cv_prog_cxx_works=no])
+                AC_LANG_POP([C++])])
+
+AM_CONDITIONAL([CXX_WORKS], [test "x$rw_cv_prog_cxx_works" = "xyes"])
```

7.4.2 Integrating patches found on the Web

When integrating a patch of which you are not the author, you have to add a few things in the header of the patch itself.

Depending on whether the patch has been obtained from the project repository itself, or from somewhere on the web, add one of the following tags:

```
Backported from: <some commit id>
```

or

```
Fetched from: <some url>
```

It is also sensible to add a few words about any changes to the patch that may have been necessary.

7.4.3 Upstreaming patches

OpenADK tries to avoid any patches to the source code. If a patch could not be avoided, it should be tried to make the patch of a good quality to get it upstream. OpenADK tries to report any found issues and try to send in any upstream compatible patches.

7.5 Debugging OpenADK

To analyze any build problems, use verbose output:

```
$ make v
```

To analyze any inter package dependency problems, use make debug output:

```
$ make --debug=b
```

If you have a problem with a specific package, use following command to capture the output and send it to the OpenADK developer:

```
$ make package=<pkgname> clean package > pkgname.log 2>&1
```

Chapter 8

Legal notice and licensing

8.1 Complying with open source licenses

All of the end products of OpenADK (toolchain, root filesystem, kernel, bootloaders) contain open source software, released under various licenses.

Using open source software gives you the freedom to build rich embedded systems, choosing from a wide range of packages, but also imposes some obligations that you must know and honour. Some licenses require you to publish the license text in the documentation of your product. Others require you to redistribute the source code of the software to those that receive your product.

The exact requirements of each license are documented in each package, and it is your responsibility (or that of your legal office) to comply with those requirements.

8.2 Complying with the OpenADK license

OpenADK itself is an open source software, released under the GNU General Public License, version 2 or (at your option) any later version. However, being a build system, it is not normally part of the end product: if you develop the root filesystem, kernel, bootloader or toolchain for a device, the code of OpenADK is only present on the development machine, not in the device storage.

Nevertheless, the general view of the OpenADK developer is that you should release the OpenADK source code along with the source code of other packages when releasing a product that contains GPL-licensed software. This is because the GNU GPL defines the "*complete source code*" for an executable work as "*all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable*". OpenADK is part of the *scripts used to control compilation and installation of the executable*, and as such it is considered part of the material that must be redistributed.

Keep in mind that this is only the OpenADK developer opinion, and you should consult your legal department or lawyer in case of any doubt.

Chapter 9

Contributing to OpenADK

If you want to contribute to OpenADK, you will need a git view of the project. Refer to Section 2.2 to get it.

You can either subscribe to the mailing list dev@openadk.org or send an email directly to wbx@openadk.org.

9.1 Submitting patches

When your changes are done, and committed in your local git view, *rebase* your development branch on top of the upstream tree before generating the patch set. To do so, run:

```
$ git fetch --all --tags
$ git rebase origin/master
```

Here, you are ready to generate then submit your patch set.

To generate it, run:

```
$ git format-patch -M -n -s origin/master
```

This will generate patch files automatically adding the `Signed-off-by` line.

Once patch files are generated, you can review/edit the commit message before submitting them using your favorite text editor.

Lastly, send/submit your patch set to the OpenADK developers:

```
$ git send-email --to dev@openadk.org *.patch
```

Note that `git` should be configured to use your mail account. To configure `git`, see `man git-send-email` or google it.

Make sure posted **patches are not line-wrapped**, otherwise they cannot easily be applied. In such a case, fix your e-mail client, or better, use `git send-email` to send your patches.

9.1.1 Cover letter

If you want to present the whole patch set in a separate mail, add `--cover-letter` to the `git format-patch` command (see `man git-format-patch` for further information). This will generate a template for an introduction e-mail to your patch series.

A *cover letter* may be useful to introduce the changes you propose in the following cases:

- large number of commits in the series;
- deep impact of the changes in the rest of the project;

- RFC [1];
- whenever you feel it will help presenting your work, your choices, the review process, etc.

9.2 Reporting issues/bugs, get help

Try to think as if you were trying to help someone else; in that case, what would you need?

Here is a short list of details to provide in such case:

- host machine (OS/release)
- git version of OpenADK
- target for which the build fails
- package(s) which the build fails
- the command that fails and its output
- the make.log file, generated when make v is used
- any information you think that may be relevant

Additionally, you can add the `.config` file.

[1] RFC: (Request for comments) change proposal

Chapter 10

Appendix

10.1 Network configuration

10.1.1 loopback devices

Example for loopback device configuration:

```
auto lo
iface lo inet loopback
```

10.1.2 static network configuration

Example for an ethernet network card:

```
auto eth0
iface eth0 inet static
  address 192.168.1.1
  netmask 255.255.255.0
  broadcast +
  gateway 192.168.1.254
```

The DNS resolver must be manually configured in /etc/resolv.conf. The plus for the broadcast value, will calculate the correct broadcast address for the network.

10.1.3 dynamic network configuration

Example for an ethernet network card:

```
auto eth0
iface eth0 inet dhcp
```

10.1.4 bridge configuration

Example for a network bridge with two ethernet network interfaces and an ip address:

```
auto br0
iface br0 inet static
        address 192.168.99.1
        netmask 255.255.255.0
        broadcast +
        bridge-ports eth0 eth1
```

Just a bridge without an ip address:

```
auto br0
iface br0 inet manual
         bridge-ports eth0 eth1
```

You need to install either Busybox brctl applet or the bridge-utils package. The required kernel modules will be automatically selected.

10.1.5 VLAN network interfaces

Example configuration of a network interface with VLAN ID 8 without any ip configuration:

```
auto eth0.8
iface eth0.8 inet manual
```

You need to install Busybox vconfig applet. The required kernel modules will be automatically selected.

10.1.6 PPP over Ethernet

Typical DSL configuration:

```
auto ppp0
iface ppp0 inet ppp
        use-template pppoe
        provider isp
        ppp-mtu 1412
        ppp-username foo
        ppp-password bar
        ppp-device eth1
```

The provider can be used as argument for "pon" and "poff" commands. You need to install the ppp and ppp-mod-pppoe package. The required kernel modules will be automatically selected.

10.1.7 wireless client configuration

Example wireless client configuration, secured with WPA2:

```
auto wlan0
iface wlan0 inet dhcp
        wireless-ssid myap
        wireless-channel 11
        wireless-mode sta
        wireless-security wpa2
        wireless-passphrase xxxxxx
```

You need to install iw and wpa_supplicant packages. For older wireless drivers you need to install wireless-tools instead of iw and use the following variable to choose the right tools:

```
wireless-extension 1
```

10.1.8 wireless accesspoint configuration

To configure an access point use following example:

```
auto wlan0
iface wlan0 inet static
        address 192.168.40.10
        netmask 255.255.255.0
        broadcast +
        wireless-ssid myap
        wireless-channel 8
        wireless-mode ap
        wireless-security wpa2
        wireless-passphrase xxxxxx
```

You need to install hostapd and iw/wireless-tools packages.

10.1.9 hso umts modem

If you have a HSO UMTS modem, you can use following to configure internet access:

```
auto hso0
iface hso0 inet manual
        pin 1234
        apn your.apn
```

10.1.10 ATM configuration

For example a configuration on a Linksys AG241 router with integrated DSL modem, you can configure two ATM devices to distinguish between Internet and IPTV traffic:

```
auto eth0.1
iface eth0.1 inet manual

auto eth0.8
iface eth0.8 inet manual

auto nas0
iface nas0 inet manual

auto nas1
iface nas1 inet manual
        atm-vpi 1
        atm-vci 34

auto br0
iface br0 inet manual
        bridge-ports eth0.1 nas0

auto br1
iface br1 inet manual
        bridge-ports eth0.8 nas1
```

More network setups can be implemented on request.