

## CHAPTER 2



# Native Development

There are projects in this book that require a bit more than the simple usage of Arduino reference APIs. Especially Chapter 7, where the OpenCV and V4L2 are explained, and some examples will not run as simple sketches (Arduino programs). In these cases, you will need to know how to make a new build, how to create the toolchains specifically to your computer and how to use the cross-compilers to compile native applications.

The build system used to create the Intel Galileo images—the Yocto project—is very powerful, but it is not as straightforward as a simple `make` command. There is some computer preparation that must be done. It is also good to know a little bit about how the build system works and how to compile the SPI and SD card images for Intel Galileo and Intel Quark toolchains in order to have the cross-compilers in hand.

This chapter also shows you how to build a very simple native Hello World application after the installation of the toolchains.

Considering that you will be able to create your own releases, especially the SPI images, there is some instruction on how to recover your board—in case you make a mistake and brick your board.

In the end, this chapter brings you some knowledge that will be necessary in the next chapters. It is not the intention to bring a full understanding of all techniques involved in a native development, especially debugging.

## Introduction to the Yocto Build System

Suppose that you are creating a great product that uses Linux as an embedded operating system because it is open source and free—reducing the product costs, and brings a great operating system to your users. Does this sound right? The answer is that it depends, because Linux is an amazing operating system, but “reducing the cost” in an embedded development could be a really huge nightmare if you do not have good control of the features required by your product, such as which Linux distribution is able to meet your requirements with minimal effort to join all the pieces together.

In order to create a custom Linux image and bring exactly the features that you need, the Yocto project was created to be a flexible and customizable platform for different hardware architectures and code.

The Yocto project brings a series of tools, methods, and code—allowing you to choose the CPU that your product targets, the software and hardware components, and the footprint size—to build a software release based in the Linux operating system. Among the CPU supported are the Intel Architecture (IA), ARM, PowerPC, and MIPS.

Besides the product releases, Yocto also allows you to build tools like system development kits (SDK) and applications to be used with your product. For example, with Intel Galileo boards, we'll cover how to build your own toolchain that contains cross-compilers for different operating systems.

Once the Yocto project is established with the configuration and components, when new components must be added or removed, or even if a new product must be created based in a legacy one that is supported by Yocto, everything will be easier because your product is reusable.

The Yocto project is maintained by the Linux Foundation, meaning that your product will be independent of any vendor or company. Companies like Intel, Dell, Mindspeed, Wind River, Mentor Graphics, Panasonic, and Texas Instruments, among others, participate in the Yocto project.

## Yocto and this Book

To understand all the details regarding the Yocto project, another book dedicated exclusively to Yocto would be necessary, because in the same manner that Yocto is powerful, it is extensive—with a lot of details involved.

In this chapter specifically, some basic concepts regarding Yocto are explained so that you understand the build process in an Intel Galileo and Intel Quark context.

Instead of executing a bunch of commands to have your builds done without any idea of what is going on, this section brings a minimal overview about how the build process works and what the messages on your computer monitor that appear during the build mean.

If you are interested in understanding Yocto more deeply, it is recommended that you access the Yocto's documentation at <https://www.yoctoproject.org/documentation> and the manual at <http://www.yoctoproject.org/docs/current/ref-manual/ref-manual.html>.

## Poky

**Poky** is the name given to the build system in a Yocto project. Poky depends on a task executor and scheduler called the *bitbake* tool.

**Bitbake** executes all the steps of the build process, based in a group of configuration files and metadata. Basically, bitbake parses and runs several shell scripts and the Python code running all the compilations. If you are a regular C/C++ developer, you usually have dependences of makefiles that were processed having the compilers invoked when you ran the good old make command. Imagine that you have a complex project with different software components and you need to run the make for each of them. Bitbake in a Yocto project context might be considered the “global make command,” but you will definitely not use any make commands because bitbake will invoke all of them for you.

The metadata defines which components to build, the components version, and how to build each of them. The metadata can be broken into three individual parts:

- **Configuration files:** Bitbake based in configuration files (.conf) that holds the global definition of variables, the compilation flags, where libraries and applications must be placed, the machine architecture to be used, and so forth.
- **Bitbake classes:** The bitbake classes, is also known as *bbclasses*, are defined in a file with the .bbclass extension. Mostly, the heavy things during the build are done with the definitions in these files, like how the RPM packages are generated, how the root file system is created, and so forth.
- **Recipes:** The recipes are the files with .bb extensions and define the individual pieces of the software to be built, the packages that must be included, where and how to obtain source code and patches, the dependencies, which features and definitions you want to enable in a source, and others.

Perhaps these definitions sound a little complicated for a build system, but they are the magic key to making the system flexible, even if it sounds overengineered. However, in order to better understand how Poky works, and the build system in general, let's build an Intel Galileo image and discuss step by step what's going on during the procedure.

Figure 2-1 was created by a Yocto project team and represents the Yocto build system flow.

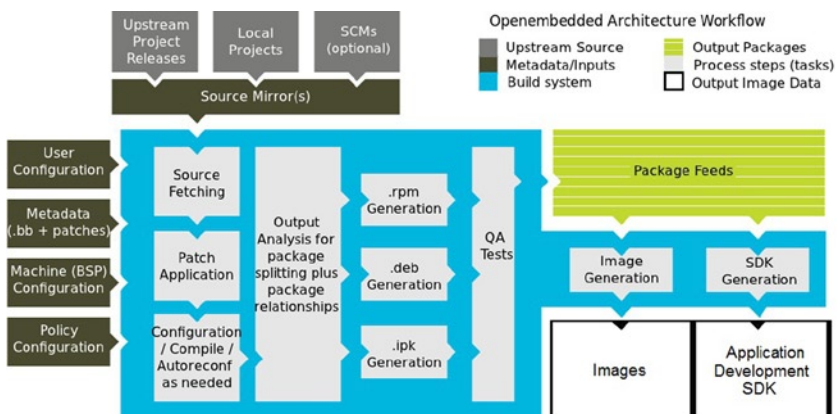


Figure 2-1. The Yocto build system flow

Figure 2-1 shows how the Yocto build process works. It warrants a few paragraphs to explain each step.

Along with input files and data, the user (**User Configuration**), policy (**Policy Configuration**), and machine configurations (**Machine BSP Configuration**) are loaded and the metadata files are parsed (**Metadata .bb + patches**).

The build process starts downloading the components from remote repositories, fetching local packages, HTTPS, web sites, FTP sites, and so on, as shown in the **Source Mirror(s)**.

Once all the necessary code is fetched to the working area (**Source Fetching**), the necessary patches are applied (**Path Application**) and the configurations are applied (**Configuration/Compile/Autoreconf**) based on the information retrieved from the input files and data.

Then thousands of software code starts to compile and the output files goes to a staging area (**output analysis**) until the expected packages are created (`.rpm`, `.deb`, or `.ipk`). You will use the **IPK** files in this book.

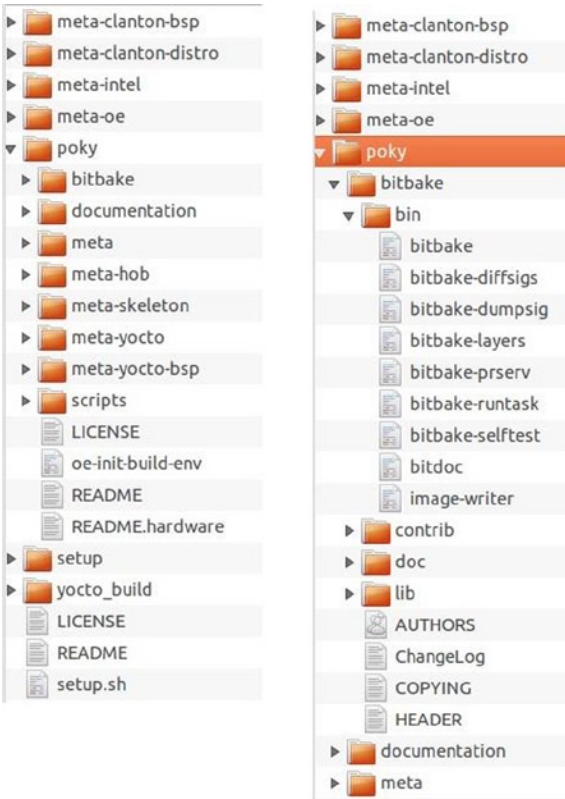
Some sanity tests are done during the generation of output files (**QA tests**) until all the necessary packages are created and fed (**package feeds**) to generate the final output images (**Image and Application Development SDK**).

Note that you will need an Internet connection, because lots of code will be downloaded to complete the build process.

## The Build System Tree at a Glance

In the next section you will learn how to download metafiles and Poky to build Intel Galileo images and your toolchain. Before building and executing a series of instructions, it would be interesting to have an overview of how the files are organized in the Poky tree and the Intel Galileo metafiles.

Figure 2-2 (left) shows the code structure that you will see when you download the code necessary to build an Intel Galileo and the toolchain.



**Figure 2-2.** *The Poky and the layers (left) and bitbake tool (right)*

As you can see, there is a folder called poky that contains the basic structure of a Yocto build system. For example, in the poky directory there is a bitbake directory that contains the **bitbake binary tool** and other utilities, as shown in Figure 2-2 (right), as well as some directories starting with meta\* prefix. Each meta\* directory is, in fact, a **layer** containing metadata—in other words, recipes, classes, and configuration files.

On top of the poky directory are other layers, like meta-clanton-bsp, meta-clanton-distro, meta-intel, and meta-oe, which, of course, have their respective recipes, classes, and configuration files, as well as any other metadata.

What defines which layers will in fact be part of compilation is a file called `bblayers.conf` in the `yocto_build/conf` directory shown in Listing 2-1.

**Listing 2-1.** bblayers.conf

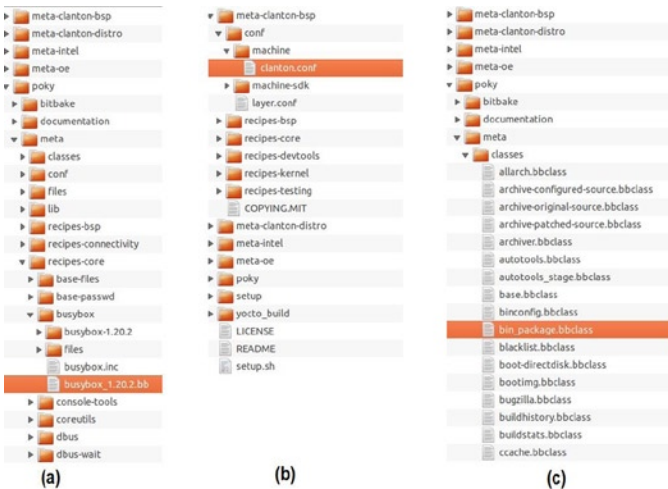
```
# LAYER_CONF_VERSION is increased each time build/conf/bblayers.conf
# changes incompatibly
LCONF_VERSION = "6"

BBPATH = "${TOPDIR}"
BBFILES ?= ""
BBLAYERS ?= " \
  /home/mcramon/BSP_1.0.4_T/meta-clanton_v1.0.1/poky/meta \
  /home/mcramon/BSP_1.0.4_T/meta-clanton_v1.0.1/poky/meta-yocto \
  /home/mcramon/BSP_1.0.4_T/meta-clanton_v1.0.1/poky/meta-yocto-bsp \
  /home/mcramon/BSP_1.0.4_T/meta-clanton_v1.0.1/meta-intel \
  /home/mcramon/BSP_1.0.4_T/meta-clanton_v1.0.1/meta-oe/meta-oe \
  /home/mcramon/BSP_1.0.4_T/meta-clanton_v1.0.1/meta-clanton-distro \
  /home/mcramon/BSP_1.0.4_T/meta-clanton_v1.0.1/meta-clanton-bsp \
"
```

It is time to explore the tree a bit more and check out recipe, configuration, and class files.

## An Example of a Recipe (.bb)

Let's look at a recipe file, choose the valid layers, and search for one. For example, let's suppose you chose the meta layer; if you explore this layer a little, you will find very interesting recipes, like the `busybox_1.20.2.bb` recipe shown in Figure 2-3(a).



**Figure 2-3.** Examples of recipe (a), configuration (b), and class files (c)

Open the busybox recipe and you will see a code structure similar to the one shown in Listing 2-2.

**Listing 2-2.** busybox\_1.20.2.bb

```
require busybox.inc
PR = "r7"

SRC_URI = "http://www.busybox.net/downloads/busybox-${PV}.tar.
bz2;name=tarball \
    file://B921600.patch \
    file://get_header_tar.patch \
    file://busybox-appletlib-dependency.patch \
    file://run-parts.in.usr-bin.patch \
    file://watch.in.usr-bin.patch \
...
...
...

file://inetd"

SRC_URI[tarball.md5sum] = "e025414bc6cd79579cc7a32a45d3ae1c"
SRC_URI[tarball.sha256sum] =
"eb13ff01dae5618ead2ef6f92ba879e9e0390f9583bd545d8789d27cf39b6882"

EXTRA_OEMAKE += "V=1 ARCH=${TARGET_ARCH} CROSS_COMPILE=${TARGET_PREFIX}
SKIP_STRIP=y"
```

Note that the recipes contain a SRC\_URI variable that defines the URLs to download busybox, and respective md5 and sha256 checksum to make sure that the package downloaded was the one expected. **The** EXTRA\_OEMAKE only adds compilation flags during the build.

Each recipe is parsed and the Yocto build process assumes some functions during the build processing. Some functions are listed next; they can be customized or simply excluded according to the configurations:

- do\_fetch
- do\_unpack
- do\_patch
- do\_configure
- do\_compile
- do\_install
- do\_package

Each function is related to the Yocto build flow, as explained earlier, and when you run the build, you will see these functions displayed on your computer, so you will have an idea of what the build process stage is for each package.

The whole list of functions that might be executed during the compilation is specified in Chapter 8 of **the Yocto Project Reference Manual** at <http://www.yoctoproject.org/docs/1.7/ref-manual/ref-manual.html#ref-tasks>.

## An Example of a Configuration File (.conf)

At this point, you have a good idea about what a recipe is, so let's look at an example of a configuration file.

Configuration files are usually under a folder called `conf` in a layer. A good example is the configuration filename `clanton.conf` that belongs to the `meta-clanton-bsp` layer under the `/conf/machine` folder. The content of this file is shown in Listing 2-3 and Figure 2-3(b).

### Listing 2-3. `clanton.conf`

```

#@TYPE: Machine
#@NAME: clanton

#@DESCRIPTION: Machine configuration for clanton systems

PREFERRED_PROVIDER_virtual/kernel ?= "linux-yocto-clanton"
PREFERRED_VERSION_linux-yocto-clanton ?= "3.8"

require conf/machine/include/ia32-base.inc
include conf/machine/include/tune-i586.inc

#Avoid pulling in GRUB
MACHINE_ESSENTIAL_EXTRA_RDEPENDS = ""

MACHINE_FEATURES = "efi usb pci"

SERIAL_CONSOLE = "115200 ttyS1"
#SERIAL_CONSOLES = "115200;ttyS0 115200;ttyS1"

EXTRA_IMAGEDEPENDS = "grub"
PREFERRED_VERSION_grub = "0.97+git%"

```

In this configuration file, you can see the definitions for `clanton` machines, such as the serial port speed and the TTY devices to be used as serial console, the kernel name and version, and the drivers supported, like EFI, USB, and PCI.



## An Example of a Class File (.bbclass)

The third example of a metadata component is the class file. They keep under a folder called `classes` with a `.bbclass` extension. As an example, using the meta layer, search for the class file `bin_package.bbclass`, as in Listing 2-4 and shown in Figure 2-3(c).

**Listing 2-4.** `bin_package.bbclass`

```
#
# ex:ts=4:sw=4:sts=4:et
# -*- tab-width: 4; c-basic-offset: 4; indent-tabs-mode: nil -*-
#
# Common variable and task for the binary package recipe.
# Basic principle:
# * The files have been unpacked to ${S} by base.bbclass
# * Skip do_configure and do_compile
# * Use do_install to install the files to ${D}
#
# Note:
# The "subdir" parameter in the SRC_URI is useful when the input package
# is rpm, ipk, deb and so on, for example:
#
# SRC_URI = "http://foo.com/foo-1.0-r1.i586.rpm;subdir=foo-1.0"
#
# Then the files would be unpacked to ${WORKDIR}/foo-1.0, otherwise
# they would be in ${WORKDIR}.
#

# Skip the unwanted steps
do_configure[noexec] = "1"
do_compile[noexec] = "1"

# Install the files to ${D}
bin_package_do_install () {
    # Do it carefully
    [ -d "${S}" ] || exit 1
    cd ${S} || exit 1
    tar --no-same-owner --exclude='./patches' --exclude='./.pc' -cpf - . \
        | tar --no-same-owner -xpf - -C ${D}
}

FILES_${PN} = "/"

EXPORT_FUNCTIONS do_install
```

This `bbcClass` file provides information on what must be done for all meta layers that make usage of binary package recipes, and in this cases *skips* the `configure` (`do_configure`) and `compile` (`do_compile`) procedures indexing `noexec` to 1, but takes action during the package installation (`do_install`).

## Creating Your Own Intel Galileo Images

After a small introduction on how the Yocto build system works, it is time to create your own releases using Poky. It is essential to prepare your computer to run the build system, because a series of requirements are necessary to make the build system functional.

### Preparing Your Computer

The first thing to do is prepare your computer to be able to build. With Yocto, the system basically runs on Linux, but if you have Windows or Mac OSX, I really recommend that you install a virtual machine, such as VMWare (<http://www.vmware.com>) or Oracle Virtual Box (<https://www.virtualbox.org>), and install one of the following Linux distributions:

- Ubuntu 12.04 (LTS)
- Ubuntu 13.10
- Ubuntu 14.04 (LTS)
- Fedora release 19 (Schrödinger's Cat)
- Fedora release 20 (Heisenbug)
- CentOS release 6.4
- CentOS release 6.5
- Debian GNU/Linux 7.0 (Wheezy)
- Debian GNU/Linux 7.1 (Wheezy)
- Debian GNU/Linux 7.2 (Wheezy)
- Debian GNU/Linux 7.3 (Wheezy)
- Debian GNU/Linux 7.4 (Wheezy)
- Debian GNU/Linux 7.5 (Wheezy)
- Debian GNU/Linux 7.6 (Wheezy)
- openSUSE 12.2
- openSUSE 12.3
- openSUSE 13.1

If you have a recent version of a Linux operating system, but it is not listed in the previous distribution list, it is recommended to check that this list is not outdated. To check the most recent distributions supported by Yocto, read the “Supported Linux Distribution” section in the Yocto Reference Project Manual at <http://www.yoctoproject.org/docs/1.7/ref-manual/ref-manual.html#detailed-supported-distros>.

Some people are able to compile successfully on Mac OSX, but there are so many steps necessary to make this possible that having a virtual machine is the quickest solution.

This book shows a complete build process using Linux **Ubuntu 12.04.04**. If you have a computer or a virtual machine with Ubuntu installed, and you want to check the version, you can run the following command in a terminal shell:

```
mcramon@ubuntu: ~/ $ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description: Ubuntu 12.04.4 LTS
Release: 12.04
Codename: precise
```

The next step requires the installation of some packages used by bitbake during the build process. The easiest way to install all the dependences is to run the following command:

```
mcramon@ubuntu:~/ $ sudo apt-get install subversion libcurl4-openssl-dev
uuid-dev autoconf texinfo libssl-dev libtool iasl bitbake diffstat gawk
chrpath openjdk-7-jdk connect-proxy autopoint p7zip-full build-essential
gcc-multilib vim-common gawk wget git-core
```

There is an important note regarding the IASL that is a compiler used to support ACPI (Advanced Configuration and Power Interface). When Intel included support to run Windows on Intel Galileo boards, a new power management configuration was created, and, consequently, the IASL compiler had to be updated to attend the ACPI 5.0 specification. Thus, when you install the **IASL** (one of the components) in the previous command, you need to make sure it supports **ACPI revision 5.0** or greater.

If you are using Ubuntu 14, the repositories already point to a version of IASL that supported ACPI 5.0; however, if you have Ubuntu 12, you will probably have problems, because the repositories point to the IASL version that only supports ACPI revision 4.0 and will have problems compiling the UEFI packages. So, if you have Ubuntu 12, the easiest way to install the correct IASL without upgrading your OS or pointing to the repositories of version 14 is to install from a source with the following commands:

```
mcramon@ubuntu:~/tools$ sudo apt-get remove iasl
mcramon@ubuntu:~/tools$ sudo apt-get install libbison-dev flex
mcramon@ubuntu:~/tools$ mkdir iasl
mcramon@ubuntu:~/tools$ cd iasl/
mcramon@ubuntu:~/tools/iasl$ git clone git://github.com/acpica/acpica.git
mcramon@ubuntu:~/tools$ cd acpica
mcramon@ubuntu:~/tools/acpica$ make
```

After the make command compiles and links everything, the output files will be in the `.../generate/unix/bin` folder.

```
mcramon@ubuntu:~/tools/acpica$ cd ./generate/unix/bin
mcramon@ubuntu:~/tools/iasl/acpica/generate/unix/bin$ ./iasl
Intel ACPI Component Architecture
ASL+ Optimizing Compiler version 20141107-64 [Dec 12 2014]
Copyright (c) 2000 - 2014 Intel Corporation
```

### Supports **ACPI Specification Revision 5.1**

The previous command gives you an installation of IASL that supports ACPI 5.1 and, of course, it is enough to meet the requirements of ACPI 5.0 because it points to the latest release of IASL repositories in <https://github.com/acpica/acpica>.

Next, just create a link in `/usr/bin/iasl` pointing to your IASL, compiled manually:

```
mcramon@ubuntu:~/sudo ln -s <YOUR IASL PATH> /usr/bin/iasl
```

For example:

```
mcramon@ubuntu:~/sudo ln -s /home/mcramon/tools/iasl/acpica/generate/unix/
bin/iasl /usr/bin/iasl
```

After you install all the packages, your machine is able to run the Yocto builds and you need to follow some steps to create your images, as commented in the next section.

## The SPI vs. SD Card Images

The Intel Galileo images are based in Linux 3.8 and there are two possible images (targets): the SPI image or the SD card image.

The SPI image is an image that fits on Intel Galileo SPI flash memory. It contains the very basic software, but allows running the sketches (Arduino programs) and contains some basic utilities, like busybox.

The SD card image must be stored in a micro SD card with a maximum capacity of 32GB and that allows booting Intel Galileo from it. This image contains a powerful variety of software, such as Python, node.js, and the drivers to support Intel WiFi and Bluetooth cards, among others.

Both images have the same procedure to build, changing only the target name in the bitbake command. However, with the SD images, you just need to copy some of the build output files in the micro SD cards; on the other hand, the SPI images require additional steps and can be created as capsule files or binary files, which will be discussed later.

The next sections explain how to build Intel Galileo and toolchain images.

## Building Intel Galileo Images

There are some steps that must be followed in order to prepare all the metafiles necessary to build such images. The instructions shown in this chapter are related to release 1.0.4. (I can guarantee that the process for 1.0.5 will be simpler because you will not need to worry about downloading and applying patches manually. So, if you are reading this book and a release newer than 1.0.4 is available, you will not need to follow all of these steps, especially the manual application of patches.) The steps are outlined here:

1. **Create a directory where your build will be placed.**

```
mcramon@ubuntu:~/ $ mkdir BSP_1.0.4_build
```

2. **Download the BSP patches.** Access the download center at [https://downloadcenter.intel.com/Detail\\_Desc.aspx?DwnldID=24355](https://downloadcenter.intel.com/Detail_Desc.aspx?DwnldID=24355) and read the instructions on how to compile the BSP. With 1.0.4, there are instructions to access the GitHub link at <https://github.com/01org/Galileo-Runtime> and download the file <https://github.com/01org/Galileo-Runtime/archive/1.0.4.tar.gz>. Next, decompress the downloaded file:

```
mcramon@ubuntu:~/ $ wget https://github.com/01org/Galileo-Runtime/archive/1.0.4.tar.gz
mcramon@ubuntu:~/ $ tar -xf Galileo-Runtime-1.0.4.tar.gz
mcramon@ubuntu:~/ $ cd Galileo-Runtime-1.0.4
```

3. **Decompress the patches.**

```
mcramon@ubuntu:~/ $ tar -xvf patches_v1.0.4.tar.gz
patches_v1.0.4/
patches_v1.0.4/.DS_Store
patches_v1.0.4/meta-clanton.patches/
patches_v1.0.4/. _patch.meta-clanton.sh
patches_v1.0.4/patch.meta-clanton.sh
patches_v1.0.4/. _patch.Quark_EDKII.sh
patches_v1.0.4/patch.Quark_EDKII.sh
patches_v1.0.4/. _patch.sysimage.sh
patches_v1.0.4/patch.sysimage.sh
patches_v1.0.4/Quark_EDKII.patches/
patches_v1.0.4/sysimage.patches/
patches_v1.0.4/sysimage.patches/.DS_Store
patches_v1.0.4/sysimage.patches/sysimage_v1.0.1+1.0.4.patch
patches_v1.0.4/Quark_EDKII.patches/.DS_Store
patches_v1.0.4/Quark_EDKII.patches/Quark_EDKII_v1.0.2+ACPI_for_
Windows.patch
```

```

patches_v1.0.4/meta-clanton.patches/.DS_Store
patches_v1.0.4/meta-clanton.patches/meta-clanton.post-patch.init.
patch
patches_v1.0.4/meta-clanton.patches/meta-clanton_v1.0.1+quark-
init.patch
patches_v1.0.4/meta-clanton.patches/post-setup.patches/
patches_v1.0.4/meta-clanton.patches/post-setup.patches/.DS_Store
patches_v1.0.4/meta-clanton.patches/post-setup.patches/1.usb_
improv_patch-1.patch
patches_v1.0.4/meta-clanton.patches/post-setup.patches/2.GAL-193-
cloader-1.patch
patches_v1.0.4/meta-clanton.patches/post-setup.patches/3.GAL-199-
start_spi_upgrade-1.patch
patches_v1.0.4/meta-clanton.patches/post-setup.patches/4.MAKER-
222-Sketch_download_unstable-5.patch
patches_v1.0.4/meta-clanton.patches/post-setup.patches/GAL-118-
USBDeviceResetOnSUSRES-2.patch
patches_v1.0.4/meta-clanton.patches/post-setup.patches/patch.sh
patches_v1.0.4/meta-clanton.patches/post-setup.patches/uart-
1.0.patch
patches_v1.0.4/meta-clanton.patches/post-setup.patches/uart-
reverse-8.patch

```

If you pay attention to the files extracted, you will realize that there are patches for several different repositories, such as meta-clanton, UEFI firmware, and native BSP code. The usage of these patches will be discussed according to the steps to build the BSP or to generate the Intel Galileo Images.

#### 4. Extract the meta-clanton.

At this point you have some tar.gz files extracted in your directory, such as the directory for SPI flash tools, the firmware based on Intel EDKII, and the sysimage templates; but what really matters at this point is the meta-clanton directory that must be decompressed.

```

mcramon@ubuntu:~/ $ tar -zxvf meta-clanton_v1.0.1.tar.gz
Galileo-Runtime-1.0.4/
Galileo-Runtime-1.0.4/Quark_EDKII_v1.0.2.tar.gz
Galileo-Runtime-1.0.4/README.txt
Galileo-Runtime-1.0.4/grub-legacy_5775f32a+v1.0.1.tar.gz
Galileo-Runtime-1.0.4/meta-clanton_v1.0.1.tar.gz
Galileo-Runtime-1.0.4/patches_v1.0.4.tar.gz
Galileo-Runtime-1.0.4/quark_linux_v3.8.7+v1.0.1.tar.gz
Galileo-Runtime-1.0.4/spi-flash-tools_v1.0.1.tar.gz

```

Alternatively, you can decompress all files, if you want, by running the following command:

```
mcramon@ubuntu:~/ $ for file in $(ls *.tar.gz); do tar -zxvf
$file;done
```

Enter the decompressed meta-clanton directory, and then observe the files and directories that you have.

```
mcramon@ubuntu:~/ $ cd meta-clanton_v1.0.1/
mcramon@ubuntu:~/ $ ls
LICENSE meta-clanton-bsp meta-clanton-distro README setup
setup.sh
```

Note that you have the meta-clanton layer, but the main build processor Poky is not present and you need to fetch it.

5. **Apply the meta-clanton patches.** Return to the previous directory and run the meta-clanton patches with following command:

```
mcramon@ubuntu:~/ $ cd ..
mcramon@ubuntu:~/ $ ./patches_v1.0.4/patch.meta-clanton.sh
```

This patch fetches new metafiles and the Poky, and then applies code patches.

Internally, the `patch.meta-clanton.sh` script calls a second script named `setup.sh` that downloads the some new metafiles that are included in the meta-clanton directory. The new metafiles are meta-intel and meta-oe. Also, two new directories were prepared: poky and yocto\_build. This might take some time, depending on the speed of your Internet connection.

You can check the new files as follows:

```
mcramon@ubuntu:~/ $ cd meta-clanton_v1.0.1/
mcramon@ubuntu:~/ $ ls
LICENSE meta-clanton-bsp meta-clanton-distro meta-intel
meta-oe poky README setup setup.sh yocto_build
```

At times during the Intel Galileo development, new bugs arise and new fixes are introduced. The Intel Galileo BSP images lays on Intel Clanton BSP baseline but the development of two lines run in parallel with some merges of Intel Galileo fixes sporadically. When new fixes arises before any official Intel Clanton baseline release, then patches are provided and the Intel Galileo BSP continues independently.

For example, this chapter is based on release 1.0.4, but when you downloaded the BSP sources, you have notices such as `meta-clanton_v1.0.1.tar.gz` that mean baseline **1.0.1**. In this case, Intel provides patches that must be applied on top of 1.0.1, and once applied, you have a legitimate source 1.0.4. It is great if there is no patch to be applied, because the baseline is in sync with previous Intel Galileo fixes.

So, the second action done by `patch.meta-clanton.sh` is to apply not only code fixes but also possible recipe files that must be correct; for example, patching a new OpenSSL code or applying security fixes.

Intel is doing its best to replace the patches script with efficient `.bbappend` files provided as a source, so that you will not need to apply any patches manually.

6. **Set the environment variables.** After applying all the patches, it is necessary to set the environment variables and Poky directly where the build should start. To do this, run the following commands:

```
mcramon@ubuntu:~/ $ cd ./meta-clanton_v1.0.1
mcramon@ubuntu:~/ $ source poky/oe-init-build-env yocto_build
```

At the end of this command, your prompt in the terminal shell will be automatically moved to the `yocto_build` directory.

7. **Enable the cache and set the number of threads.** This step is just a recommendation. It is not necessary to follow because you could start your build; however, these changes will enable the cache and might make your build a little bit faster.

Open the file `../meta-clanton/yocto_build/conf/local.conf` with the text editor of your preference.

The change is the variable `BB_NUMBER_THREADS` that represents the maximum number of threads that your bitbake command will be able to handle. My suggestion is to multiply the numbers of threads on your computer processor by 2; for example, if your computer supports 8 threads, you can change this number to 16. If you are using a free version of virtual machines, check the number of core processors that it allows you to set. For example, the free version of VMware only allows setting a maximum of four cores, and if each core of your processor holds one single thread, then `BB_NUMBER_THREADS` could be 8.

```
BB_NUMBER_THREADS = "12"
```

Still, in `yocto.conf` you can make the following changes:

```
SSTATE_DIR ?= "/tmp/yocto_cache-sstate"
SOURCE_MIRROR_URL ?= "file:///tmp/yocto_cache/"
INHERIT += "own-mirrors"
BB_GENERATE_MIRROR_TARBALLS = "1"
```

By enabling the cache, if your build is interrupted for some reason, such as a lapse moment of Internet disconnection, if you re-execute the bitbake command, the build will not start from scratch because the cache is reused and the code that was previously downloaded does not need to be downloaded again.

The next step is the compilation itself.



8. **Compile the images.** It is time to execute the build process using the bitbake tool. At this point, you have two possible images related to Intel Galileo: the **SPI image** and the **SD card image**. To check the name of each target release, type the command `bitbake -s`, which brings all the targets supported by the current configurations:

```
mcramon@ubuntu:~$ bitbake -s |grep galileo
galileo-target                               :0.1-r0
image-full-galileo                          :1.0-r0
image-spi-galileo                          :1.0-r0
```

The target `image-full-galileo` creates the SD card image; `image-spi-galileo` creates the SPI image; and `galileo-target` must be ignored because it is not used anymore.

Then, using bitbake again, you can run the build process for the target you want to work. For example, for SPI you just need to run this:

```
mcramon@ubuntu:~$ bitbake image-spi-galileo
```

All the configurations are checked; the download of the sources, packages, and patches that compose the software is started; each component is properly set, enabling and disabling features and software definitions; and finally, everything is compiled and the images are generated.

During the compilation process, you will be able to see the `do_` actions in place of different recipes, the number of tasks completed and to be completed, and warnings if the mirrors failed to download the expected code. You do not need to worry about warnings, because they are an indication that the code failed to be fetched and a different mirror will be used. You only need to worry if there are errors reported, and in this case, you need to identify the recipe file and check whether the URL mirrors changed, which would fix the file, or if you have a generic error like an Internet connection loss or insufficient space in the device.

Figure 2-4 shows a snapshot of a full image process given after the command `bitbake image-full-galileo` is used to create SD card releases. Note that there are 2,924 tasks to be done, but only 190 were executed, which means that this is the beginning of the compilation. You can also observe some actions in place, such as `do_configure`, `do_compile`, `do_patch`, and `do_unpack`, for different recipes.

```

File Edit View Search Terminal Help
NOTE: consider defining a PREFERRED_PROVIDER entry to match glibc-gconv-ibm862-native
NOTE: multiple providers are available for runtime glibc-gconv-ibm863-native (eglibc-locale, nativesdk-eglibc-locale)
NOTE: consider defining a PREFERRED_PROVIDER entry to match glibc-gconv-ibm863-native
NOTE: multiple providers are available for runtime glibc-gconv-ibm865-native (eglibc-locale, nativesdk-eglibc-locale)
NOTE: consider defining a PREFERRED_PROVIDER entry to match glibc-gconv-ibm865-native
NOTE: multiple providers are available for runtime glibc-gconv-ibm866-native (eglibc-locale, nativesdk-eglibc-locale)
NOTE: consider defining a PREFERRED_PROVIDER entry to match glibc-gconv-ibm866-native
NOTE: multiple providers are available for runtime glibc-gconv-ibm869-native (eglibc-locale, nativesdk-eglibc-locale)
NOTE: consider defining a PREFERRED_PROVIDER entry to match glibc-gconv-ibm869-native
NOTE: multiple providers are available for jpeg (jpeg, libjpeg-turbo)
NOTE: consider defining a PREFERRED_PROVIDER entry to match jpeg
NOTE: Preparing runqueue
NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks
WARNING: Failed to fetch URL http://www.zlib.net/zlib-1.2.7.tar.bz2, attempting MIRRORS if available
WARNING: Failed to fetch URL ftp://ftp.ossp.org/pkg/lib/uuid/uuid-1.6.2.tar.gz, attempting MIRRORS if available
Currently 11 running tasks (190 of 2924):
0: bison-native-2.7-r1 do_configure (pid 36419)
1: gmp-native-5.1.0-r2 do_configure (pid 38487)
2: openssl-native-1.0.1h-r15.2 do_compile (pid 53868)
3: linux-libc-headers-3.8-r0 do_unpack (pid 61560)
4: icu-native-50.1.2-r0 do_configure (pid 62887)
5: gcc-cross-initial-4.7.2-r20 do_unpack (pid 6896)
6: elfutils-native-0.148-r11 do_compile (pid 7941)
7: binutils-native-2.23.1-r3 do_unpack (pid 9287)
8: do-native-5.3.21-r0 do_unpack (pid 13552)
9: gmp-native-5.1.0-r2 do_compile (pid 39250)
10: ossp-uuid-native-1.6.2-r2 do_patch (pid 19313)

```

Figure 2-4. bitbake output for full image compilation

If the compilation is fine, the next step is to check the output files.

9. In the end, if everything downloads and is configured, compiled, and linked, and patches are applied, you should have the images available in the `...meta-clanton/yocto_build/tmp/deploy/images` directory.

If you created SD card images, you just need to copy the files to your SD card; otherwise, some additional steps are necessary with SPI images.

The next section explains how to build the toolchain, but if you are excited to test your release, read the “*Booting Intel Galileo with Your Own Images*” section in this chapter.

## Building and Using the Cross-Compiler Toolchain

It is important to understand how to create the **cross-compilers** and **IPK** packages because some chapters of this book will make use of them, especially in Chapter 7, and, of course, if you want to create native applications.

The next sections explain how to build the toolchain and how you might generate a toolchain for different operating systems.

Note that if your intention is only to create images to Intel Galileo boards, then this section is not mandatory.

## Compiling the Toolchain for Different Architectures

If you are a Windows or Mac OSX user, you are probably running the Yocto build using a virtual machine. At this point, you might be asking if you can create a toolchain for your native operating system, instead of using virtual machines for everything, including the toolchain.

The answer is yes, and it is very simple to create a toolchain for other architectures, even if you have a Linux machine, because it is one of the proposals of the Yocto build system.

To make such a change, it is necessary to open the file `.../meta-clanton/yocto_build/conf/local.conf` and add the variable `SDKMACHINE` followed by a string that describes the machine architecture designed to the SDK build.

```
SDKMACHINE = "i386-darwin"

BB_NUMBER_THREADS = "12"

PARALLEL_MAKE = "-j 14"

MACHINE = "clanton"
DISTRO ?= "clanton-tiny"
EXTRA_IMAGE_FEATURES = "debug-tweaks"
USER_CLASSES ?= "buildstats image-mklibs image-prelink"
PATCHRESOLVE = "noop"
CONF_VERSION = "1"
```

Basically, the strings for different operating systems are shown in Table 2-1.

**Table 2-1.** Machine Architecture Definition

String	Target Architecture
i586	Linux, 32-bit
x86_64	Linux, 64-bit
i386-darwin	OSX
i686-mingw32	Windows, 32- and 64-bit

If the `SDKMACHINE` is not explicitly declared, then the toolchain will assume the computer architecture that runs the Yocto build.

You need to use the text editor of your preference, or simply change the machine using a command line. For example, if you want to specify the target as 32-bit Linux, you can run the following:

```
mcramon@ubuntu:~/ $ cd meta-clanton_v1.0.1/yocto_build
mcramon@ubuntu:~/ $ echo 'SDKMACHINE = "i586"' >> conf/local.conf
```

The next sections discuss how to build and install the toolchains for different operating systems.

## Building the Toolchains

The generations of toolchains require the same steps mentioned in the “*Building Intel Galileo Images*” section; however, the `bitbake` command is different and additional layers must be downloaded.

Note that it is always recommended to check any possible changes in the process—how the toolchains are generated in case this book becomes outdated. In this case, consult the *Quark BSP Build Guide*, which you can access at <http://www.intel.com/content/dam/www/public/us/en/documents/guides/galileo-quark-x1000-bsp-guide.pdf>.

The instructions in this section generate the toolchain based on the `uclibc` library, because it is the default library set in the metafiles. If you are interested in creating the toolchains based in `eglibc`, you need to read Chapter 7, specifically the “Preparing the BSP Software Image and Toolchain” section.

The generation of toolchains is different for Linux, Windows, and OSX, as you will read in the following instructions.

### Linux

The following is the command to generate the toolchain for 32-bit Linux:

```
mcramon@ubuntu:~//$ cd ./meta-clanton_v1.0.1
mcramon@ubuntu:~//$ source poky/oe-init-build-env yocto_build
mcramon@ubuntu:~//$ echo 'SDKMACHINE = "i586"' >> conf/local.conf
mcramon@ubuntu:~//$ bitbake meta-toolchain
```

If you want to generate for 64-bit Linux, you need to change the `SDKMACHINE` to `x86_64`. Alternatively, you can replace the command `bitbake meta-toolchain` with `bitbake image_full -c populate_sdk` and the result will be the same.

### OSX

OSX requires you have a legitimate Mac computer with **OSX 10.8** or later and with **Xcode 5.1.0** or later installed. Initially, using your Mac computer, perform the following steps:

1. Go to the App Store and install **Xcode 5.1.0** or later.
2. Install the command-line development tools using **Preferences ► Downloads** and choose command-line tools.
3. Using the terminal shell, create the file `OSX-sdk.zip` with following commands:

```
$ mkdir ~/Desktop/OSX-sdk
$ cd ~/Desktop/OSX-sdk
$ ditto `xcrun --sdk macosx10.8 --show-sdk-path` .
$ cd ..
$ zip -yr OSX-sdk OSX-sdk
```

4. Copy the `OSX-sdk.zip` to a directory in your Linux virtual machine. The following are the commands to create the OSX toolchain:

```
mcramon@ubuntu:~/~$ cd ./meta-clanton_v1.0.1
mcramon@ubuntu:~/~$ sed -i 's|setup/gitsetup.py -c setup/$1.cfg -w
mcramon@ubuntu:~/~$ sed -i 's|setup/gitsetup.py -c setup/$1.cfg -w
$1|setup/gitsetup.py -c setup/$1.cfg -w $1 --depth=1|' setup/
gitsetup.py mcramon@ubuntu:~/~$ ./setup.sh
mcramon@ubuntu:~/~$ git clone git://git.yoctoproject.org/meta-darwin
mcramon@ubuntu:~/~$ cd meta-darwin ; git checkout
03b7dd85732838d78e4879332b1cc005dae25754 ; cd ..
mcramon@ubuntu:~/~$ (cd poky && patch -p1) < meta-darwin/oecore.patch
mcramon@ubuntu:~/~$ mv <YOUR DIRECTORY HERE>/OSX-sdk.zip meta-
darwin/recipes-devtools/osx-runtime/files darwinpath="$(pwd)/
meta-darwin"
mcramon@ubuntu:~/~$ echo 'SDKMACHINE = "i386-darwin"' >> yocto_
build/conf/local.conf
mcramon@ubuntu:~/~$ echo "BBLAYERS += \"\$darwinpath\"" >> yocto_
build/conf/bblayers.conf
mcramon@ubuntu:~/~$ source poky/oe-init-build-env yocto_build
mcramon@ubuntu:~/~$ bitbake meta-toolchain
```

## Windows

For Windows, the commands are the same for Windows, 64 or 32 bits; however, a sequence of two bitbakes is required in addition to the extra metafiles.

```
mcramon@ubuntu:~/~$ sed -i 's|setup/gitsetup.py -c setup/$1.cfg -w $1|setup/
gitsetup.py -c setup/$1.cfg -w $1 --depth=1|' setup/gitsetup.py
mcramon@ubuntu:~/~$ ./setup.sh
```

```
mcramon@ubuntu:~/~$ git clone -b dylan git://git.yoctoproject.org/meta-mingw
mcramon@ubuntu:~/~$ (cd poky && patch -p1) < meta-mingw/oecore.patch
```

```
mcramon@ubuntu:~/~$ mingwpath="$(pwd)/meta-mingw"
mcramon@ubuntu:~/~$ echo 'SDKMACHINE = "i686-mingw32"' >> yocto_build/conf/
local.conf
mcramon@ubuntu:~/~$ echo "BBLAYERS += \"\$mingwpath\"" >> yocto_build/conf/
bblayers.conf
```

```
mcramon@ubuntu:~/~$ cd $WORKSPACE/meta-clanton_v1.0.1poky
mcramon@ubuntu:~/~$ wget http://git.yoctoproject.org/cgi/cgit.cgi/poky/patch/
meta/classes/sstate.bbclass?id=4273aa4287ecd36529f2d752c76ab8d09afc33c3 -O
sstate.bbclass.patch
git am sstate.bbclass.patch
```

```
mcramon@ubuntu:~/ $ cd $WORKSPACE/meta-clanton_v1.0.1
mcramon@ubuntu:~/ $ source poky/oe-init-build-env yocto_build
mcramon@ubuntu:~/ $ bitbake gcc-crosssdk-initial -c cleansstate
mcramon@ubuntu:~/ $ bitbake meta-toolchain
```

## The Output Files

The output files will be in the `.../meta-clanton_v1.0.1/yocto_build/tmp/ deploy/ sdk` directory, while the ipk packages will be in the directory `.../meta-clanton_v1.0.1/ yocto_build/tmp/ deploy/ ipk` directory.

The output filename depends on whether your computer is 32 or 64 bits, the architecture that the toolchain is designated for (we will discuss later), and the `uclibc` or `eglibc` library that the image is based on. In the end, you will have just a single script file; however, it is a big file at around 260MB.

For example, if you compile in a 64-bit Linux machine with an Intel processor, the output filename is `clanton-tiny-uclibc-x86_64-i586-toolchain-1.4.2.sh`.

The next sections discuss how to install and test the toolchains.

## Installing the Cross-Compilers

The installation of the toolchain just requires you to execute the script created and choose a destination folder, as shown:

```
mcramon@ubuntu:~/toolchain$ ./clanton-tiny-uclibc-x86_64-i586-toolchain-1.4.2.sh
Enter target directory for SDK (default: /opt/clanton-tiny/1.4.2):
You are about to install the SDK to "/opt/clanton-tiny/1.4.2". Proceed[Y/n]?Y
[sudo] password for mcramon:
Extracting SDK...done
Setting it up...done
SDK has been successfully set up and is ready to be used.
```

The shell script inflates the toolchain in the directory chosen, and all programs that make part of the toolchain are promptly accommodated.

The “**Creating a Hello World!**” section in this chapter brings a practical usage of the toolchain.

## Creating a Hello World!

This section requires you to have built and properly installed the toolchain in your computer.

If you enter the toolchain directory chosen during the installation, you will notice many binary files, including the compilers and directories, but initially what matters is a file that starts with `environment-setup-*`; for example, in my setup I have the file named as `environment-setup-i586-poky-linux-uclibc`.

This file contains a lot of variables—such as CC, CXX, CPP, AR, and NM—that must be exported to your computer shell. They are used to compile, link, and archive your native programs with the toolchain, so primarily you need to make this variable part of the development environment, sourcing it as follows:

```
mcramon@ubuntu:/opt/clanton-tiny/1.4.2$ source environment-setup-i586-poky-linux-uclibc
```

For example, you will be able to compile a program with `$(CC) -c $(CFLAGS) $(CPPFLAGS)` since CC points to the C compilers, CFLAGS to the C compiler flags, and CPPFLAGS to the C++ compiler flags. If you check some of these variables after sourcing them, you will see something like this:

```
mcramon@ubuntu:/opt/clanton-tiny/1.4.2$ echo $CC
i586-poky-linux-uclibc-gcc -m32 -march=i586 --sysroot=/opt/clanton-tiny/1.4.2/sysroots/i586-poky-linux-uclibc
mcramon@ubuntu:/opt/clanton-tiny/1.4.2$ echo $CFLAGS
-O2 -pipe -g -feliminate-unused-debug-types
mcramon@ubuntu:/opt/clanton-tiny/1.4.2$ echo $CXXFLAGS
-O2 -pipe -g -feliminate-unused-debug-types -fpermissive
```

Listing 2-5 brings a simple Hello World program written in C, which is present in the code folder of this chapter.

*Listing 2-5.* HelloWorld.c

```
#include <stdio.h>
int main(int argc, char const* argv[])
{
    printf("Hello, World! This is Intel Galileo!\n");
    return 0;
}
```

Copy this program to your computer and compile it using the variables you exported.

```
mcramon@ubuntu:/ ${CC} ${CFLAGS} HelloWorld.c -o HelloWorld
```

You should have the executable HelloWorld created using the cross-compiler. Just copy this file to a **micro SD card** formatted using FAT or FAT32. If you do not know how to format the micro SD card, read the “*Boot from SD Card Image*” section of this chapter for instructions.

Insert the micro SD card on your Intel Galileo and boot the board connecting the power supply. Also connect the serial cables, as explained in Chapter 1, and open a Linux terminal shell.

Then locate the micro SD card mounted to the `/media/mmcblk0p1` partition, and execute the `HelloWorld`, as shown:

```
root@clanton:/# cd /media/mmcblk0p1/
root@clanton:/media/mmcblk0p1# ls
HelloWorld
root@clanton:/media/mmcblk0p1# ./HelloWorld
Hello, World! This is Intel Galileo!
```

If you see the output message, it means that your toolchain is functional and generating the binaries correctly. There are multiples ways to transfer your executable to the board, using either WiFi, Ethernet, a pen drive, or a micro SD card. For more information, read the “*Transferring Files Between Intel Galileo and Computers*” section in Chapter 5.

You can also create a simple makefile for this `HelloWorld` by simply using the variables exported by the `environment-setup-i586-poky-linux-uclibc` as a base. For example, Listing 2-6 shows a makefile for the `HelloWorld` program.

**Listing 2-6.** Makefile

```
SHELL = /bin/bash
TARGET_NAME = i586-poky-linux-uclibc
DIST = clanton-tiny
CC = $(TARGET_NAME)-gcc -m32 -march=i586 --sysroot=/opt/$(DIST)/1.4.2/sysroots/$(TARGET_NAME)
CFLAGS = -O2 -pipe -g -feliminate-unused-debug-types
OUTPUT_FILE = HelloWorld

all: target

target: $(patsubst %.c,%.o,$(wildcard *.c))
          $(CC) $(CFLAGS) $^ -o $(OUTPUT_FILE)

clean:
          rm -f $(TARGET_BIN) *.o $(OUTPUT_FILE)
```

The makefile created is designated to target `i586-poky-linux-uclibc`, as stored in the variable `TARGET_NAME` and considers the toolchain installed in the `/opt/clanton-tiny` directory according to the `CC` variable. So, if you create the toolchain for a different target, or used a different directory installation, it is necessary to adapt this makefile.

The makefile also brings three commands: `clean` to clean all the object files and the output file named as `HelloWorld`, because it is the value in the `OUTPUT_FILE` variable; `all` and `target` do the same thing—in other words, compile the C programs, invoking the compiler pointed by `CC` and `CFLAGS`.

To create a `HelloWorld`, all you need to do is type `make`.



```
mcramon@ubuntu:~/native$ make
i586-poky-linux-uclibc-gcc -m32 -march=i586 --sysroot=/opt/clanton-tiny/1.4.2/sysroots/i586-poky-linux-uclibc -O2 -pipe -g -feliminate-unused-debug-types -c -o HelloWorld.o HelloWorld.c
i586-poky-linux-uclibc-gcc -m32 -march=i586 --sysroot=/opt/clanton-tiny/1.4.2/sysroots/i586-poky-linux-uclibc -O2 -pipe -g -feliminate-unused-debug-types HelloWorld.o -o HelloWorld
```

The next section talks about debugging native applications.

## Debugging Native Applications

It is possible to debug native application and kernel modules using GDB, Eclipse, and JTAG tools. This book focuses on Arduino projects, so all debugging methods are concentrated in the Intel Arduino IDE, and not in native applications or kernel contexts. In the scope of this book, it is important to know how build systems work, how to build and compile native applications, and how to generate the cross-compilers, because these features will be used in the following chapters, especially when you work with OpenCV and V4L2 in Chapter 7.

However, if you are interested in learning how to debug native applications, Intel provides a very good tutorial about how to use Eclipse with Intel Galileo and Intel Edison in the developer zone. This tutorial can be accessed at <https://software.intel.com/en-us/getting-started-for-c-c-plus-plus-eclipse-galileo-and-edison>.

For kernel debugging, GDB, and JTAG enabling using openOCD, it is recommended that you read the *Source Level Debugging using OpenOCD/GDB/Eclipse on Intel Quark SoC X1000* manual, present in the manuals folder of this chapter, or you can access it at <https://communities.intel.com/docs/DOC-22203>.

The next section explains how to make Intel Galileo boot with the images that you created with poky.

## Booting Intel Galileo with Your Own Images

As explained earlier, you have two image targets related to Intel Galileo—the SPI card image and the SD card image. The procedures to make Intel Galileo boot using these images differ and must be followed as directed in the following sections.

### Booting from SD Card Images

The SD card release only requires that you copy some of the output files to a micro SD card, insert it in Intel Galileo, and then power-on the board.

### Preparing the Micro SD Card

Before copying the files, there are important details to know regarding the format of the micro SD card, which must be FAT or FAT32 with a single partition.

First of all, you need to format the SD card in your computer. Nowadays, computers offer SD card slots; if you insert the micro SD card into a SD card adaptor connected to the slot, you are able to read, write, or format your micro SD card, as shown in Figure 2-5. However, if your computer does not provide any kind of access to the micro SD card, then you will need a micro SD card reader that connects to a USB port. Figure 2-6 shows an example of an SD card reader in a laptop and a micro SD card reader device.



**Figure 2-5.** An SD card adaptor to be used with a computer

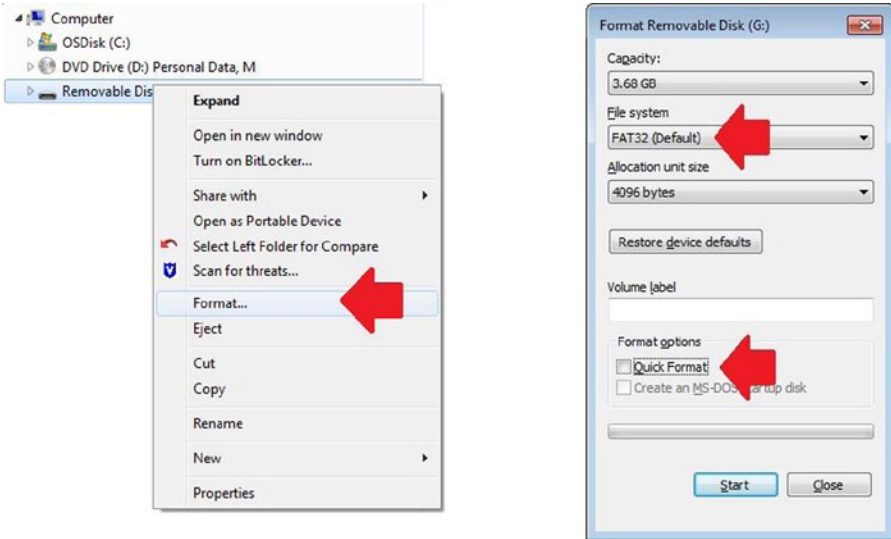


**Figure 2-6.** A micro SD card USB adaptor

With a physical connection between the micro SD card and your computer established, you just need to format the micro SD card according to your OS.

## Windows

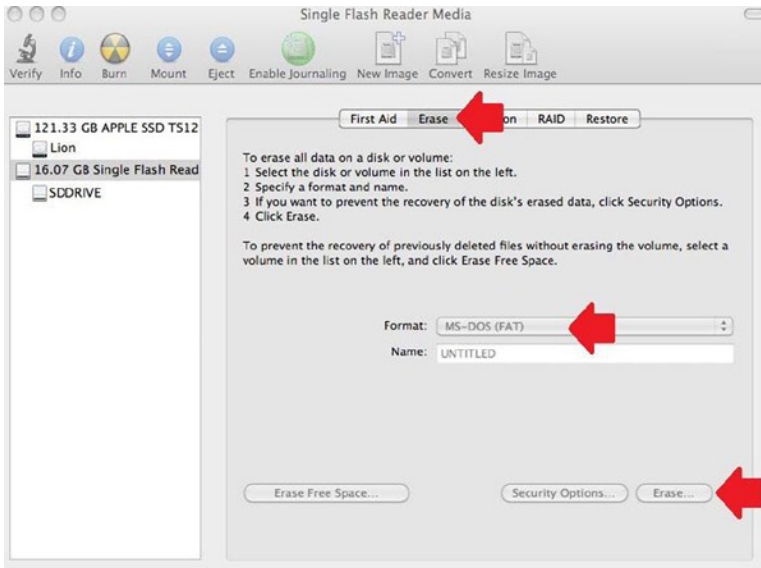
If you are a Windows user and you are running a virtual machine with Linux to run Yocto builds, you might be excited to use the regular format procedure offered by Windows; in other words, open Explorer, right-click the SD card drive, and select the Format option. In this case, deselect Quick Format and choose the right partition format, as shown in Figure 2-7.



**Figure 2-7.** Formatting the micro SD card on Windows

## Mac OSX

Formatting a micro SD card on Mac OSX is quite easy. In Spotlight, type **Disk Utility** and run the **Disk Utility** software. Click the micro SD card in the left panel and then click the Erase tab. Select the format type in the Format combo box and click the Erase button. Figure 2-8 shows the Disk Utility options.



**Figure 2-8.** The Disk Utility on Mac OS X formatting the micro SD card

## Ubuntu

On Ubuntu there are several utilities with a very nice graphical interface that can format the micro SD card, including **GParted** (<http://gparted.sourceforge.net>) and **Disk Utility** for Ubuntu (<https://apps.ubuntu.com/cat/applications/precise/gnome-disk-utility/>). To avoid any new software installation, however, it is possible to format using simple command-line commands. The steps are as follows:

1. Open a terminal by pressing **Ctrl+Alt+T** at same time.
2. Type the command `df` to check the partition in your computer, including the micro SD card mounted. Then identify the device name that defines the micro SD card; for example, `/dev/sdb1`.
3. Unmount the SD card using the `umount` command followed by the device name. For example:

```
umount /dev/sdb1
```

4. Use the `MKDOSFS` utility to format the card. For example:

```
mkdosfs -F 32 -v /dev/sdb1
```

With the micro SD card ready, it is time to copy your image into it.

## Copying Files to a Micro SD Card

If you successfully create your SD card image, enter `../yocto_build/tmp/deploy/images` in the directory and type the `ls -l` command:

```
mcramon@ubuntu $ cd ../tmp/deploy/images/
mcramon@ubuntu $ ls -l
total 150576
drwxr-xr-x 3 mcramon mcramon    4096 Nov 18 23:01 boot
-rw-r--r-- 2 mcramon mcramon   373760 Nov 19 00:04 bootia32.efi
lrwxrwxrwx 2 mcramon mcramon     42 Nov 18 23:58 bzImage -> bzImage--3.8-
r0-clanton-20141119062948.bin
-rw-r--r-- 2 mcramon mcramon  1984512 Nov 18 23:58 bzImage--3.8-r0-clanton-
20141119062948.bin
lrwxrwxrwx 2 mcramon mcramon     42 Nov 18 23:58 bzImage-clanton.bin ->
bzImage--3.8-r0-clanton-20141119062948.bin
-rw-r--r-- 1 mcramon mcramon  1689687 Nov 19 00:08 core-image-minimal-
initramfs-clanton-20141119062948.rootfs.cpio.gz
lrwxrwxrwx 1 mcramon mcramon     66 Nov 19 00:08 core-image-minimal-
initramfs-clanton.cpio.gz -> core-image-minimal-initramfs-clanton-
20141119062948.rootfs.cpio.gz
-rw-r--r-- 2 mcramon mcramon   279670 Nov 18 23:59 grub.efi
-rw-r--r-- 1 mcramon mcramon 314572800 Nov 19 00:26 image-full-galileo-
clanton-20141119062948.rootfs.ext3
lrwxrwxrwx 1 mcramon mcramon     53 Nov 19 00:26 image-full-galileo-
clanton.ext3 -> image-full-galileo-clanton-20141119062948.rootfs.ext3
-rw-rw-r-- 2 mcramon mcramon  1556960 Nov 18 23:58 modules--3.8-r0-clanton-
20141119062948.tgz
-rw-rw-r-- 2 mcramon mcramon    294 Nov 19 00:25 README_-_DO_NOT_DELETE_
FILES_IN_THIS_DIRECTORY.txt
```

There is a folder called `boot` with files and links. The only function of the links is to make “easy reading” of the files that receive a timestamp in their names. For example, the `bzImage--3.8-r0-clanton-20141119062948.bin`, where `20141119062948` is only the timestamp; thus, if you run the bitbake again without any modification, you will have another `bzImage` file with a different timestamp and a link pointing to the newest one.

Thus, you will need to copy to your micro SD card as follows:

1. `boot` (the whole directory, including subdirectories)
2. `bzImage`
3. `core-image-minimal-initramfs-clanton.cpio.gz`
4. `grub.efi`
5. `image-full-galileo-clanton.ext3`

Copy these files and directories to your micro SD card, insert it into the micro SD card slot (see Chapter 1), and power-on your Intel Galileo.

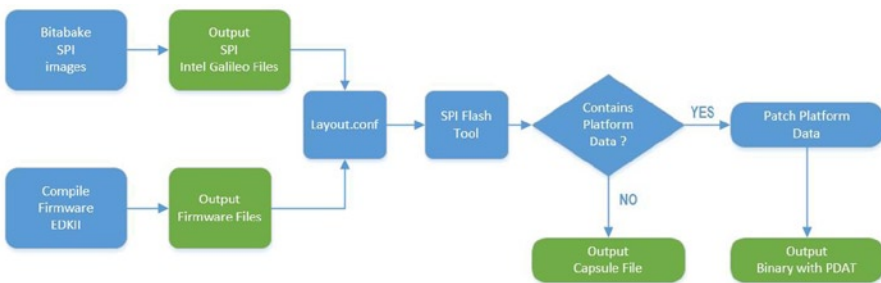
This is everything you need if you are using SD card images. The next section explains this procedure when SPI images are used.

## Booting from SPI Card Images

When you build an SPI image, the process results in a single file created to store your image in the SPI flash memory. In this case, two different types of files can be built:

- **Capsule file:** This file contains the system images, the kernel, the file system partition, and the boot loader packages (grub), but it does not contain the platform data. Platform data informs the MAC address of your Ethernet controller and the board model, such as Intel Galileo or Intel Galileo Gen2. This file is very useful in most cases; if you have a board without boot issues and the Ethernet controller is working with a correct MAC address, the file is very handy. Usually, capsule files (or cap files) contain the `.cap` extension, and you can flash Intel Galileo boards using the Intel Arduino IDE or the UEFI shell, which will be discussed later.
- **Binary file:** This binary file contains everything; in other words, everything in a capsule file, plus the platform data. Usually, these files have the `.bin` extension and must be flashed with an SPI programmer.

Figure 2-9 shows a flowchart that explains the process to generate both of these files.



**Figure 2-9.** SPI files generation flowchart

Initially it is necessary to generate the Intel Galileo SPI images that will generate the SPI files as output.

In parallel, it is possible to compile the firmware and generate the files related to firmware as output. Then a template is mounted using a file named `layout.conf` that contains all the ingredients necessary to build files a single file that will be used to flash the SPI flash memory.

With `layout.conf` ready, the SPI flash tool is called to generate capsule and binary files without platform data. If the intention is to have files without platform data, at this point the capsule files might be used; otherwise, the platform data must be patched using a Python script, which will be discussed later. A final binary with all the information is created.

## Creating the Capsule Files Flash Files

When you downloaded the BSP board support package in the step 2 of the “Creating your Own Intel Galileo Images” section of this chapter, you should have noticed that files in addition to the meta-clanton data were downloaded, among them files called `spi-flash-tools_v1.0.1.tar.gz` and `Quark_EDKII_v1.0.1.tar.gz`.

## Compiling the UEFI Firmware

To decompress the Linux kernel before your board boots, there is firmware responsible to initialize the board components, including the Intel Quark SoC. It also assumes other activities after the boot.

The Intel Galileo provides firmware compliant with UEFI (Unified Extensible Firmware Interface) standards that consist of boot procedures, runtime services calls, and data tables used for power management methods like ACPI (Advanced Configuration and Power Interface).

EDKII means the environment cross-platform for firmware development.

Of course, to understand the UEFI specification and EDKII development process, it would require a full book dedicated to this subject. In the context of this book, the concept is limited to how to build the EDKII, which is one of the core elements to have a functional SPI image.

The next sections discuss how to prepare your environment and how to compile the firmware.

## Preparing the Environment

The following are the dependences to compile the UEFI firmware:

- Python 2.6 or newer
- Any GCC and G++ with versions between 4.3 and 4.7
- Subversion
- `uuid-dev`
- IASL

If you run the command line proposed in the “*Preparing Your Computer*” section, you should be fine with these dependences, except Python. Check to see if you have Python installed on your Linux by running the following command:

```
mcramon@ubuntu:~$ dpkg --get-selections | grep -v deinstall|grep -i python
```

Or you can run this:

```
mcramon@ubuntu:~$ python --version
Python 2.7.3
```

If you do not have Python installed, you can install it by following the instructions at <https://www.python.org/downloads/>. If you want a quick try using version 2.7.6, you can run the following commands:

```
mcramon@ubuntu:~/$ wget https://www.python.org/ftp/python/2.7.6/Python-2.7.6.tgz
mcramon@ubuntu:~/$ tar -zxvf Python-2.7.6.tgz
mcramon@ubuntu:~/$ cd Python-2.7.6/
mcramon@ubuntu:~/$ ./configure
mcramon@ubuntu:~/$ make
mcramon@ubuntu:~/$ make install
```

After this, you are ready to compile the firmware by following the steps presented in the next section.

## Compiling the Firmware

Once you have downloaded the right package, you need to follow these instructions step by step:

1. **Extract the package.** The first thing to do is go back to the base directory and decompress the file:

```
mcramon@ubuntu $ tar -xvf Quark_EDKII_v1.0.2.tar.gz
```

Unfortunately, it is necessary to apply patches manually, but fortunately this can be done with a single command line:

```
mcramon@ubuntu $ ./patches_v1.0.4/patch.Quark_EDKII.sh
```

This patch only fixes some ACPI tables to support Windows, which is not within the scope of this book, but it is recommended to run this patch anyway to keep your firmware updated.

2. **Prepare the SVN project.** The EDKII is maintained using the SVN configuration control release tool.

```
mcramon@ubuntu:~/$ cd Quark_EDKII_v1.0.2/
mcramon@ubuntu:~/$ ./svn_setup.py
mcramon@ubuntu:~/$ svn update
mcramon@ubuntu:~/$ export WORKSPACE=$(pwd)
```



The first command, `./svn_setup.py`, is a Python script that brings a series of code related to EDKII to your computer. The command `svn update` makes certain that you have the latest changes in fetched files. This step might take few minutes, depending on your Internet connection speed.

3. **Identify the GCC that you have installed.** There is a *compilation flag* used during the firmware compilation that depends of the GCC compiler installed on your computer. To check which version you have, you can type the following command:

```
mcramon@ubuntu:~/ $ gcc --version
gcc (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3
Copyright (C) 2011 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
```

In the example, GCC informed us that the version is **4.6.3**, which means that the flag to be used in the compilation line of EDKII will be the string `GCC46`.

The GCC and G++ compilers tested at the time that this book was written was between version 4.3 and 4.7, which means that the flags supported are **GCC43, GCC44, GCC45, GCC46, and GCC47**.

The easiest way to keep the right flag during your compilation is to export a variable in bash with the latest character of the version:

```
mcramon@ubuntu:~/ $ export GCCVERSION=$(gcc -dumpversion | cut -c 3)
mcramon@ubuntu:~/ $ echo $GCCVERSION
4
```

So, for GCC version 4.6.3, the last character, **6**, is stored in the variable `GCCVERSION`; if the version was 4.7.x, the character **7** would be stored.

4. **Compile the firmware.** In the folder that you extracted the EDKII, you will notice a file called `quarkbuild.sh`. This file is a shell script that compiles the firmware for you with the following options:

```
quarkbuild.sh [-x32 | -d32 | -clean] [GCC43 | GCC44 | GCC45 |
GCC46 | GCC47]
[PlatformName] [-DSECURE_LD (optional)] [-DTPM_SUPPORT
(optional)]
```

These can be defined as follows:

- `-clean`: Delete the build files/folders.
- `-d32`: Create a DEBUG build.
- `-r32`: Create a RELEASE build.
- `GCC4x`: GCC flags used for this build.
- `[PlatformName]`: Name of the Platform package you want to build.
- `[-DSECURE_LD]`: Create a Secure Lockdown build (optional).
- `[-DTPM_SUPPORT]`: Create EDKII build with TPM support (optional).

---

■ **Note** This option has a one-time prerequisite described in `CryptoPkg\Library\OpenSSL\Patch-HOWTO.txt` in the EDKII directory that you downloaded and extracted.

---

So you can type:

```
mcramon@ubuntu:~/./quarkbuild.sh -r32 GCC46 QuarkPlatform
```

Or, if you exported the `GCCVERSION` variable you can run:

```
mcramon@ubuntu:~/./quarkbuild.sh -r32 GCC4$GCCVERSION QuarkPlatform
```

Several files will be compiled, taking a few minutes to finish.

The files that really matter will be in the output directory at `Quark_EDKII_v1.0.2/Build/QuarkPlatform/RELEASE_GCC46/FV/FlashModules`.

```
mcramon@ubuntu:~/BSP_1.0.4_T/Quark_EDKII_v1.0.2/Build/QuarkPlatform/RELEASE_GCC46/FV/FlashModules$ 1 -1
EDKII_BOOTROM_OVERRIDE.Fv
EDKII_BOOT_STAGE1_IMAGE1.Fv
EDKII_BOOT_STAGE1_IMAGE2.Fv
EDKII_BOOT_STAGE2_COMPACT.Fv
EDKII_BOOT_STAGE2.Fv
EDKII_BOOT_STAGE2_RECOVERY.Fv
EDKII_NVRAM.bin
EDKII_RECOVERY_IMAGE1.Fv
Flash-EDKII-missingPDAT.bin
RMU2.bin
RMU.bin
```

If you want to see some extra debug messages, especially during the boot, you can generate the debug releases using the `-d32` flag, as follows:

```
mcramon@ubuntu:~/./quarkbuild.sh -d32 GCC4$(GCCVERSION) QuarkPlatform
```

In this case, the output directory is in the `DEBUG_GCC4X` directory instead of the `RELEASE_GCC4X` directory with same files.

5. **Create symbolic links.** If you successfully compiled the firmware, you might have the following output:

```
mcramon@ubuntu:~/Quark_EDKII_v1.0.2$ cd Build/QuarkPlatform/
mcramon@ubuntu:~/BSP_1.0.4_T/Quark_EDKII_v1.0.2/Build/QuarkPlatform$ ls
DEBUG_GCC46  RELEASE_GCC46
```

The directories `DEBUG_GCC46` and `RELEASE_GCC46` are the result of a debug and release compilation using GCC compiler version 4.6. It is necessary to simplify such directories using soft links, naming them `DEBUG_GCC` and `RELEASE_GCC` only because these are the names that the system image tools will search for.

```
mcramon@ubuntu:~/ $ ln -s DEBUG_GCC46 DEBUG_GCC
mcramon@ubuntu:~/ $ ln -s RELEASE_GCC46 RELEASE_GCC
mcramon@ubuntu:~/ $ ls -l
total 8
lrwxrwxrwx 1 mcramon mcramon 11 Nov 21 20:21 DEBUG_GCC -> ../DEBUG_GCC46
lrwxrwxrwx 1 mcramon mcramon 13 Nov 21 20:21 RELEASE_GCC -> ../RELEASE_GCC46
```

If you achieve this step, **congratulations**, you are ready to generate the next step—creating the capsule files.

## Troubleshooting Compiling the Firmware

Some problems can show up during the firmware compilation, but all of them are related to your environment settings. The following lists the most common errors and explains how to resolve them.

- **Python does not fetch the code.** In this case, the first thing to do is check whether your Internet connection is working. You can try to test by opening a web browser or via a command line using a `wget` command like this:

```
mcramon@ubuntu:~/tmp$ wget --spider http://example.com
Spider mode enabled. Check if remote file exists.
--2014-11-21 19:53:29-- http://example.com/
Resolving example.com (example.com)... 93.184.216.119,
2606:2800:220:6d:26bf:1447:1097:aa7
Connecting to example.com (example.com)|93.184.216.119|:80...
connected.
```

```

HTTP request sent, awaiting response... 200 OK
Length: 1270 (1.2K) [text/html]
Remote file exists and could contain further links,
but recursion is disabled -- not retrieving.

```

If you are behind a proxy, then you need also to configure the subversion proxy settings, editing the file located in `~/.subversion/servers`. Then search for the section `[global]` and set your proxy configuration as shown in the following lines:

```

[global]
http-proxy-host = <YOUR HOST IP>
http-proxy-port = <YOUR PORT NUMBER>
http-proxy-username = <YOUR USER NAME>
http-proxy-password = <YOUR PASSWORD>

```

- *A GCC compiler not supported.* If you have a GCC compiler that is not supported, you can download and install one of the versions supported and change the link called `gcc` in the `/usr/bin` directory to point to the old one. For example:

```

mcramon@ubuntu:~/cd /usr/bin
mcramon@ubuntu:~/sudo ln -s /usr/bin/gcc-4.6 gcc

```

This file contains a tool that is used to create the cap and binary files based in your SPI images.

The procedure for the creation is quite simple, as explained next.

## Preparing layout.conf

At this point, you need make the other zipped files that you have downloaded but not used until now. So, move to the base directory and type the following command line to decompress all of them, if you have not done so yet:

```

mcramon@ubuntu:~/BSP_1.0.4_T$ tar -zxvf spi-flash-tools_v1.0.1.tar.gz
mcramon@ubuntu:~/BSP_1.0.4_T$ tar -zxvf sysimage_v1.0.1.tar.gz
mcramon@ubuntu:~/BSP_1.0.4_T$ tar -zxvf grub-legacy_5775f32a+v1.0.1.tar.gz
mcramon@ubuntu:~/BSP_1.0.4_T$ tar -zxvf quark_linux_v3.8.7+v1.0.1.tar.gz

```

Then run a script that will create symbolic links, making the folder names much simpler:

```

mcramon@ubuntu:~/BSP_1.0.4_T$ ./sysimage_v1.0.1/create_symlinks.sh
See if we can: ln -s ./spi-flash-tools_* spi-flash-tools
Found spi-flash-tools_v1.0.1
+ ln -s spi-flash-tools_v1.0.1 spi-flash-tools
See if we can: ln -s ./Quark_EDKII_* Quark_EDKII

```

**Found Quark\_EDKII\_v1.0.2**

```
+ ln -s Quark_EDKII_v1.0.2 Quark_EDKII
```

```
See if we can: ln -s ./sysimage_* sysimage
```

**Found sysimage\_v1.0.1**

```
+ ln -s sysimage_v1.0.1 sysimage
```

```
See if we can: ln -s ./meta-clanton_* meta-clanton
```

**Found meta-clanton\_v1.0.1**

```
+ ln -s meta-clanton_v1.0.1 meta-clanton
```

```
See if we can: ln -s ./quark_linux_* quark_linux
```

**Found quark\_linux\_v3.8.7+v1.0.1**

```
+ ln -s quark_linux_v3.8.7+v1.0.1 quark_linux
```

```
See if we can: ln -s ./grub-legacy_* grub-legacy
```

**Found grub-legacy\_5775f32a+v1.0.1**

```
+ ln -s grub-legacy_5775f32a+v1.0.1 grub-legacy
```

If this script does not work it is because you are executing from the wrong directory. Make sure that you are in the base folder where you download all tar.gz files.

As you can see, the script tried to find each component of the BSP source package and create symbolic links to them using common names. For example, grub-legacy\_5775f32a+v1.0.1 turns grub-legacy, and the same process is done to the other directories, as you can see if you type `ls -l` after the script execution.

```
mcramon@ubuntu:~/BSP_1.0.4_T$ ls -l
total 5292
-rw-r--r--  1 mcramon mcramon 2657072 Nov 17 23:11 board_support_package_
sources_for_intel_quark_v1.0.1.7z
-rw-r--r--  1 mcramon mcramon   30720 Nov 17 22:54 BSP-Patches-and-Build_
Instructions.1.0.4.tar
lrwxrwxrwx  1 mcramon mcramon         27 Nov 21 20:32 grub-legacy -> grub-
legacy_5775f32a+v1.0.1
drwxr-xr-x  2 mcramon mcramon   4096 May 22  2014 grub-
legacy_5775f32a+v1.0.1
-rw-rw-r--  1 mcramon mcramon 192465 May 22  2014 grub-
legacy_5775f32a+v1.0.1.tar.gz
lrwxrwxrwx  1 mcramon mcramon         19 Nov 21 20:32 meta-clanton -> meta-
clanton_v1.0.1
drwxr-xr-x  9 mcramon mcramon   4096 Nov 18 21:58 meta-clanton_v1.0.1
-rw-rw-r--  1 mcramon mcramon 517412 May 22  2014 meta-clanton_v1.0.1.tar.gz
drwxr-xr-x  2 mcramon mcramon   4096 Oct 20 13:31 patches
lrwxrwxrwx  1 mcramon mcramon         18 Nov 21 20:32 Quark_EDKII -> Quark_
EDKII_v1.0.2
drwxr-x--- 21 mcramon mcramon   4096 Nov 21 18:48 Quark_EDKII_v1.0.2
drwxrwxr-x  6 mcramon mcramon   4096 Nov 21 18:40 Quark_EDKII_v1.0.2-svn_
externals.repo
-rwxr-xr-x  1 mcramon mcramon 1502762 Nov 21 15:20 quark_edkii_v1.0.2.tar.gz
lrwxrwxrwx  1 mcramon mcramon         25 Nov 21 20:32 quark_linux -> quark_
linux_v3.8.7+v1.0.1
```

```

drwxr-xr-x  2 mcramon mcramon    4096 May 22  2014 quark_linux_v3.8.7+v1.0.1
-rw-rw-r--  1 mcramon mcramon 236544 May 22  2014 quark_linux_
v3.8.7+v1.0.1.tar.gz
-rw-rw-r--  1 mcramon mcramon    480 May 22  2014 shatsum.txt
lrwxrwxrwx  1 mcramon mcramon    22 Nov 21  20:32 spi-flash-tools ->
spi-flash-tools_v1.0.1
drwxr-xr-x  6 mcramon mcramon    4096 May 22  2014 spi-flash-tools_v1.0.1
-rw-rw-r--  1 mcramon mcramon 219559 May 22  2014 spi-flash-tools_
v1.0.1.tar.gz
lrwxrwxrwx  1 mcramon mcramon    15 Nov 21  20:32 sysimage ->
sysimage_v1.0.1
drwxr-xr-x  9 mcramon mcramon    4096 May 22  2014 sysimage_v1.0.1
-rw-rw-r--  1 mcramon mcramon   9876 May 22  2014 sysimage_v1.0.1.tar.gz
-rw-r--r--  1 mcramon mcramon   2938 Nov 18  22:14 uart-reverse-8.patch

```

The reason for this “simplification” is related to the `sysimage` directory bringing a configuration file that tells the “**ingredients**”—in other words, the files that will be used to compose the flash image and the version of the image.

For example, check the directories that you have in the `sysimage` file:

```

mcramon@ubuntu:~/BSP_1.0.4_T$ cd sysimage
mcramon@ubuntu:~/BSP_1.0.4_T/sysimage$ ls -l
total 36
drwxr-xr-x  2 mcramon mcramon 4096 May 22  2014 config
-rwxr-xr-x  1 mcramon mcramon 2496 May 22  2014 create_symlinks.sh
drwxr-xr-x  2 mcramon mcramon 4096 May 22  2014 grub
drwxr-xr-x  2 mcramon mcramon 4096 May 22  2014 inf
-rw-r--r--  1 mcramon mcramon 1488 May 22  2014 LICENSE
drwxr-xr-x  2 mcramon mcramon 4096 May 22  2014 sysimage.CP-8M-debug
drwxr-xr-x  2 mcramon mcramon 4096 May 22  2014 sysimage.CP-8M-debug-secure
drwxr-xr-x  2 mcramon mcramon 4096 May 22  2014 sysimage.CP-8M-release
drwxr-xr-x  2 mcramon mcramon 4096 May 22  2014 sysimage.CP-8M-release-secure

```

Note that there are four directories to generate a flash image with 8MB for debug and release compilation, and for unsecure and secure boots.

In each of these directories, there is a file called `layout.conf`. This file must be changed to point to the correct “ingredients” of your build and the right version number.

To make your life easier, there is a script that does the changes in all directories automatically for you, even if you do not need to change all of them. Running the script executes the following command in the base directory:

```

mcramon@ubuntu:~/BSP_1.0.4_T$ ./patches_v1.0.4/patch.sysimage.sh

```

You might ask which changes this script really makes. Let’s assume one of the directories—`sysimage.CP-8M-debug` for example—and open the `layout.conf` file with the text editor of your preference *before running* the `patch_sysimage.sh` script. `layout.conf` is shown in Listing 2-7.

**Listing 2-7.** layout.conf

```
# WARNING: this file is indirectly included in a Makefile where it
# defines Make targets and pre-requisites. As a consequence you MUST
# run "make clean" BEFORE making changes to it. Failure to do so may
# result in the make process being unable to clean files it no longer
# has references to.
```

```
[main]
size=8388608
type=global
```

```
[MFH]
version=0x1
flags=0x0
address=0xffff08000
type=mfh
```

**[Flash Image Version]**

```
type=mfh.version
meta=version
value=0x01000105
```

```
[ROM_OVERLAY]
address=0xffffe0000
item_file=../../Quark_EDKII/Build/QuarkPlatform/PLAIN/DEBUG_GCC/FV/
FlashModules/EDKII_BOOTROM_OVERRIDE.Fv
type=some_type
```

```
[signed-key-module]
address=0xffffd8000
item_file=config/SvpSignedKeyModule.bin
svn_index=0
type=some_type
in_capsule=no
```

```
# On a deployed system, the SVN area holds the last known secure
# version of each signed asset.
# TODO: generate this area by collecting the SVN from the assets
# themselves.
```

```
[svn-area]
address=0xffffd0000
item_file=config/SVNArea.bin
type=some_type
```

```
# A capsule upgrade must implement some smart logic to make sure the
# highest Security Version Number always wins (rollback protection)
in_capsule=no
```

```
[fixed_recovery_image]
address=0xffff90000
item_file=../../Quark_EDKII/Build/QuarkPlatform/PLAIN/DEBUG_GCC/FV/
FlashModules/EDKII_RECOVERY_IMAGE1.Fv
sign=yes
type=mfh.host_fw_stage1_signed
svn_index=2
# in_capsule=no
```

```
[NV_Storage]
address=0xffff30000
item_file=../../Quark_EDKII/Build/QuarkPlatform/PLAIN/DEBUG_GCC/FV/
FlashModules/EDKII_NVRAM.bin
type=some_type
```

```
[RMU]
address=0xffff00000
item_file=../../Quark_EDKII/Build/QuarkPlatform/PLAIN/DEBUG_GCC/FV/
FlashModules/RMU.bin
type=none_registered
```

```
[boot_stage1_image1]
address=0xffec0000
item_file=../../Quark_EDKII/Build/QuarkPlatform/PLAIN/DEBUG_GCC/FV/
FlashModules/EDKII_BOOT_STAGE1_IMAGE1.Fv
sign=yes
boot_index=0
type=mfh.host_fw_stage1_signed
svn_index=1
```

```
[boot_stage1_image2]
address=0xffe80000
item_file=../../Quark_EDKII/Build/QuarkPlatform/PLAIN/DEBUG_GCC/FV/
FlashModules/EDKII_BOOT_STAGE1_IMAGE2.Fv
sign=yes
boot_index=1
type=mfh.host_fw_stage1_signed
svn_index=1
```



```
[boot_stage_2_compact]
address=0xffd00000
item_file=../../Quark_EDKII/Build/QuarkPlatform/PLAIN/DEBUG_GCC/FV/
FlashModules/EDKII_BOOT_STAGE2_COMPACT.Fv
sign=yes
type=mfh.host_fw_stage2_signed
svn_index=3
```

```
[Ramdisk]
address=0xffa60000
item_file=../../meta-clanton/yocto_build/tmp/deploy/images/image-spi-
clanton.cpio.lzma
sign=yes
type=mfh.ramdisk_signed
svn_index=7
```

```
[LAYOUT.CONF_DUMP]
address=0xffc00000
type=mfh.build_information
meta=layout
```

```
[Kernel]
address=0xff852000
item_file=../../meta-clanton/yocto_build/tmp/deploy/images/bzImage
sign=yes
type=mfh.kernel_signed
svn_index=6
```

```
[grub.conf]
address=0xff851000
item_file=grub/grub-spi.conf
sign=yes
type=mfh.bootloader_conf_signed
svn_index=5
```

```
[grub]
address=0xff800000
item_file=../../meta-clanton/yocto_build/tmp/deploy/images/grub.efi
sign=yes
fvwrap=yes
guid=B43BD3E1-64D1-4744-9394-D0E1C4DE8C87
type=mfh.bootloader_signed
svn_index=4
```

As you can see, this file has sections like [main], [MFH], [Flash Image Version], [ROM OVERLAY], and so on. Each section contains data fields with respective values, but there are two sections that the script changes:

- **[Flash Image Version]:** It is recommended that you make a simple change in the **[Flash Image Version]** because it brings version 0x01000105 in the value field. This means that when you boot your board, the version read will be 01.00.01.05, or simply 1.0.1, because 05 is omitted; and considering that the release of this example is based on 1.0.4, it is recommended to change to **0x01000400**, which means 1.0.4. If you want to see the correct version number, this change is necessary.
- **[RamDisk]:** This section needs to replace the string `image-spi-clanton.cpio.lzma` with `image-spi-galileo-clanton.cpio.lzma`, because if you check the images generated in `/meta-clanton/yocto_build/tmp/deploy/images/`, the image generated is named `image-spi-galileo.cpio.lzma`. Thus, this section should be as follows:

```
[Ramdisk]
address=0xffa60000
item_file=../../meta-clanton/yocto_build/tmp/deploy/images/image-
spi-galileo-clanton.cpio.lzma
sign=yes
type=mfh.ramdisk_signed
svn_index=7
```

In general, the script also adjusts the path filenames, removing all PLAIN directories and pointing to valid paths.

The `sysimage` brings the template with the old version because the tool did not require any changes since 1.0.1, and the template comes with the same version number.

The other sections—like [NVM Storage], [RMU], [boot\_stage1\_image1], [boot\_stage1\_image2], and [boot\_stage\_2\_compact]—search for the EDKII components that you created in the previous section; but pay attention to **DEBUG**. This explains why you created the soft links in the step 5 of the “**Steps to Compile the Firmware**” section of this chapter.

The sections [Ramdisk], [LAYOUT.CONF\_DUMP], [Kernel], [grub.conf], and [grub] try to find the elements that you generated after running the Yocto build, and the directories that you decompressed and created are simple symbolic links in this section with the `create_symlinks.sh` script.

Thus, when you run the `patch_sysimage.sh` script, the changes mentioned are automatically done in the `layout.conf` files of each directory.

## Using the SPI Tool

The SPI tool is on the `spi-flash-tool` directory that you decompressed. It is used to create the capsule files and binary files with or without platform data.

In the **same directory as your** `layout.conf` file, run the following command:

```
mcramon@ubuntu:~/ $ ../../spi-flash-tools/Makefile
```

If everything runs OK, you should have generated three new files in the same directory:

- **Flash-missingPDAT.cap:** This is the expected capsule file, absent of platform data, which you can flash to your Intel Galileo.
- **Flash-missingPDAT.bin:** This is a binary file absent of platform data necessary to generate SPI images, which is discussed in the “*Creating SPI Images with Platform Files*” section.
- **FVMAIN.fv:** This file is used to recover your board if it does not boot anymore. This is discussed in the “*What to Do If Intel Galileo Bricks*” section of this chapter.

## Flashing the Capsule Files

After a long procedure and many hours creating your capsule file, it is time to test it by flashing the SPI flash memory. In fact, there are three different ways to flash, as discussed in next sections.

### Flashing the Capsule File with the Intel Arduino IDE

This is the easiest way to flash the capsule file with the current software provided at the time this book was published. You just need to copy the `Flash-missingPDAT.cap` file in a specific folder of the IDE, as explained in the “*Updating the Firmware with a Different Firmware*” section of Chapter 3. This procedure only requires usage of the regular USB data cable, which prevents copying the capsule files with a micro SD card or a USB pen driver.

## Flashing the Capsule File with a Linux Terminal Shell

The procedure described here is exactly the same thing that Intel Arduino IDE does automatically for you, sending remote commands to Intel Galileo boards. Thus if you do not want to use the IDE, then the procedure to flash your capsule file is as follows:

1. Connect the serial cabled to Intel Galileo and open a Linux terminal, as explained in the “*Preparing Your Cables*” section of Chapter 1.
2. Check which release is being used currently in your board, checking the content of the file `/sys/firmware/board_data/flash_version`. It possible to check using a Linux terminal shell and typing the following command:

```
root@clanton:~# cat /sys/firmware/board_data/flash_version
0x01000105
```

3. Copy the `Flash-missingPDAT.cap` that you created in the previous sections to a micro SD card or a pen driver properly formatted with FAT or FAT32 in a single partition, as described in the “*Booting from SD Card*” section of this chapter.
4. If your release is 0.7.5 or 0.8.0, run the following command:

```
# insmod /tmp/0.7.5/efi_capsule_update.ko
Or
# insmod /tmp/0.8.0/efi_capsule_update.ko
```

5. If your release is 0.9.0 or 1.0.0, run the following:

```
# modprobe efi_capsule_update
```

6. With newer releases, run the following:

```
# modprobe sdhci-pci
# modprobe mmc-block
# mkdir /lib/firmware
# cd /media/mmcblk0p1/
# cp Flash-missingPDAT.cap /lib/firmware/Flash-missingPDAT.cap
# echo -n Flash-missingPDAT.cap > /sys/firmware/efi_capsule/
capsule_path
# echo 1 > /sys/firmware/efi_capsule/capsule_update
# reboot
```

Make sure that you really ran the command `reboot`; otherwise, the process to update the capsule file will not work.

## Flashing the Capsule File with a UEFI Shell

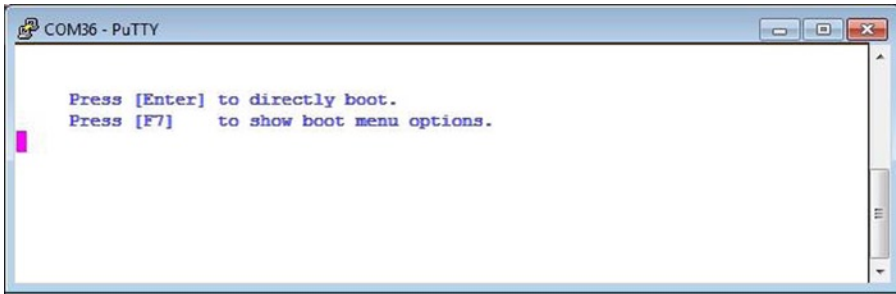
The idea on this procedure is to open a UEFI shell as soon the board boots and then flash your capsule file, but this only works if you have a board with a nonsecure boot; otherwise, the UEFI shell will be locked and this procedure will not work.

This procedure requires that you have the following:

- The `Flash-missingPDAT.cap` file that must be present in the **same directory of your** `layout.conf`.
- The `CapsuleApp.efi` file that was generated when you compiled the EDKII firmware. It must be present in the `./Quark_EDKII/Build/QuarkPlatform/PLAIN/DEBUG_GCC/FV/Applications/` directory or the `./Quark_EDKII/Build/QuarkPlatform/PLAIN/RELEASE_GCC/FV/ Applications/` directory, depending whether you compile using release or debug flags as discussed in the “*Compiling the EDKII Firmware*” section of this chapter.
- You will need serial cables to open the terminal shell, as discussed in the “*Preparing Your Cables*” section in Chapter 1. This will allow you to debug the board using a serial audio jack cable for Intel Galileo or a FTDI cable for Intel Galileo Gen 2.
- You need to know how to use some serial terminal software. Read the “*Testing Your Cables*” section in Chapter 1 to understand how to use **putty** for Windows or **minicom** for Linux or Mac OSX. However, you also need to configure the serial software to recognize special characters from your keyboard. For putty, click the left panel, **Terminal ► Keyboard**, and select the **SCO box** from **the Functions and Keys and Keypad** tab.
- Finally, you need a micro SD card or a USB pen driver.

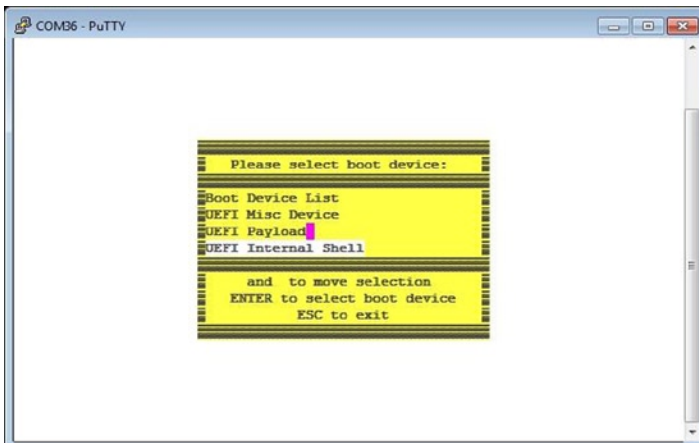
Here is the procedure that must be followed:

1. Format the micro SD card or the USB pen drive with FAT or FAT32 in a single partition, as described in the “*Booting from SD Card Images*” section of this chapter.
2. Copy the files `CapsuleApp.efi` and `Flash-missingPDAT.cap` to the micro SD card.
3. With the board off, keep the serial cable connected and open the serial terminal software, such as putty or minicom.
4. Power-on the board connecting the power supply.
5. As soon you see the image shown in Figure 2-10, press the **F7 function key**.



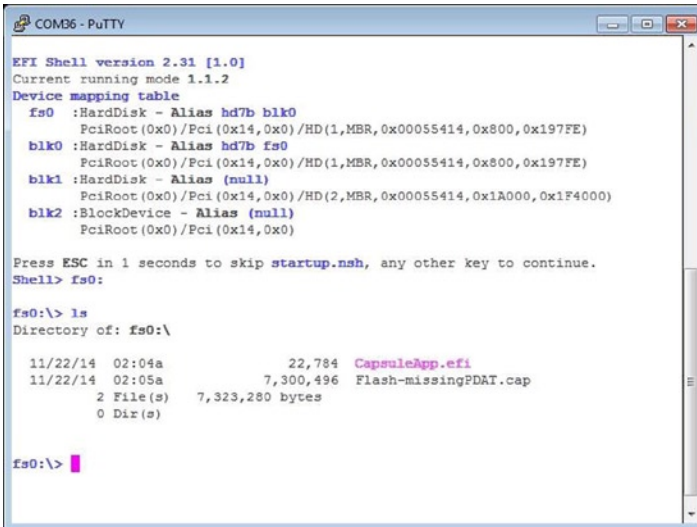
**Figure 2-10.** Initial screen just after the boot with the F7 option

- In the menu, choose the **UEFI Internal Shell** option by using the arrow keys, as shown in Figure 2-11. Press Enter and then press the ESC key to receive the shell prompt.



**Figure 2-11.** Selecting the UEFI internal shell

- You will see the partition mounted on the board. Usually, a micro SD card and a pen drive are **fs0**. Just type **fs0:** and press Enter. Check the content of the micro SD card or the USB pen driver with the command **ls**, as shown in Figure 2-12.



```

COM06 - PuTTY
EFI Shell version 2.31 [1.0]
Current running mode 1.1.2
Device mapping table
fs0 :HardDisk - Alias hd7b blk0
      PciRoot(0x0)/Pci(0x14,0x0)/HD(1,MBR,0x00055414,0x800,0x197FE)
blk0 :HardDisk - Alias hd7b fs0
      PciRoot(0x0)/Pci(0x14,0x0)/HD(1,MBR,0x00055414,0x800,0x197FE)
blk1 :HardDisk - Alias (null)
      PciRoot(0x0)/Pci(0x14,0x0)/HD(2,MBR,0x00055414,0x1A000,0x1F4000)
blk2 :BlockDevice - Alias (null)
      PciRoot(0x0)/Pci(0x14,0x0)

Press ESC in 1 seconds to skip startup.nsh, any other key to continue.
Shell> fs0:

fs0:\> ls
Directory of: fs0:\

    11/22/14  02:04a                22,784  CapsuleApp.efi
    11/22/14  02:05a            7,300,496  Flash-missingPDAT.cap
          2 File(s)      7,323,280 bytes
          0 Dir(s)

fs0:\>

```

**Figure 2-12.** Selecting the fs0 partition and checking the files

8. You should be able to see the files you copied to the micro SD card. In this case, just type the following command to start the flashing procedure.

```

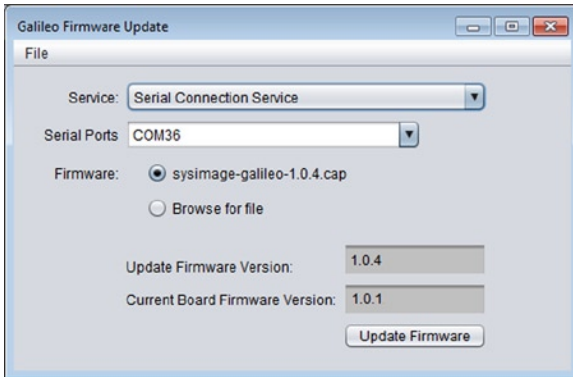
fs0:\> CapsuleApp.efi Flash-missingPDAT.cap
CapsuleApp: SecurityAuthenticateImage 0xD504410found.
CapsuleApp: creating capsule descriptors at 0xF0DE510
CapsuleApp: capsule data starts          at 0xD504410 with
size 0x6F6190
CapsuleApp: capsule block/size          0xD504410/0x6F6190

```

## Flashing the Capsule File with the Firmware Update Tool

At the end of 2014, Intel provided a new tool called the Intel Firmware Update tool that allows you to select the capsule files that come internally in the application, or to browse your desktop file system to select a custom one. This is a stand-alone application, very simple to use, and it does not require you to have the Arduino IDE installed.

Figure 2-13 shows this application's user interface.



**Figure 2-13.** Intel Galileo Firmware Update tool prototype

Subscribe to the Intel Maker Community forum at <https://communities.intel.com/community/makers> to receive more information about this tool, as many other updates will be available.

## Creating SPI Images Flash Files

Imagine this hypothetical situation: due to some mistake, you realized that you bricked your Intel Galileo and it does not boot anymore. Before ordering a new one, you can consider flashing using an SPI flash programmer, but you need to have the binary build patched with platform data files.

In previous sections, I mentioned how to compile using Yocto and how to generate the capsule and binary files that do not contain platform data.

To follow the procedures in this section, you should

1. Have successfully compiled the UEFI firmware (EDKII packages).
2. Identified the Ethernet MAC address of your board.
3. Have a flash programmer, such as DediProg.

As explained before, platform data contains information like the Ethernet MAC address of your board and which board model you have. Thus, each board should contain a unique and exclusive platform file, because each board contains an exclusive MAC address.

To discover the Ethernet MAC address of your board, you just need to take a look at the white label on your board, as shown in Figure 2-14.





**Figure 2-14.** The white label with the exclusive Ethernet MAC address

The tool responsible for generating the binary with platform data is actually a Python script named `platform-data-patch.py` in the `.../spi-flash-tools/platform-data` directory. The only thing that this script does is patch the binaries with that platform data configuration file.

In this same directory there is a platform-data template called `sample-platform-data.ini`, as shown in Listing 2-8.

**Listing 2-8.** `sample-platform-data.ini`

```
# Every module contains:
# [unique name]
# id=decimal integer, data type identifier
# desc=string, short description of a data; max 10 characters
# data.value=[ABC | CAFEBEBA | xyz abc | /path/to/file ]
# data.type=[hex.uint[8/16/32/64] | hex.string | utf8.string | file]
# ver=decimal integer, version number; if not specified defaults to 0
```

```
# WARNING: the platform type data.value MUST match the MRC data.value below
```

```
[Platform Type]
```

```
id=1
desc=PlatformID
data.type=hex.uint16
```

```
# ClantonPeak 2, KipsBay 3, CrossHill 4, ClantonHill 5, KipsBay-fabD 6, GalileoGen2 8
data.value=2
```

```
# WARNING: the MRC data.value MUST match the platform type data.value above
```

```
[Mrc Params]
```

```
id=6
ver=1
desc=MrcParams
data.type=file
data.value=MRC/clantonpeak.v1.bin
#data.value=MRC/kipsbay.v1.bin
#data.value=MRC/crosshill.v1.bin
#data.value=MRC/clantonhill.v1.bin
#data.value=MRC/kipsbay-fabD.v1.bin
#data.value=MRC/GalileoGen2.bin
```

```
# If you are developing MRC for a new system you can alternatively
# inline the value like this:
```

```
# data.type=hex.string
# data.value=00000000000000010101000300000100010101017C92000010270000102700
00409C000006
```

```
# The unique MAC address(es) owned by each device are typically found
# on a sticker. You must find it(them) and change the bogus values
# below.
```

```
[MAC address 0]
```

```
id=3
desc=1st MAC
data.type=hex.string
data.value=FFFFFFFFF00
```

```
[MAC address 1]
```

```
id=4
desc=2nd MAC
data.type=hex.string
data.value=02FFFFFFFFF01
```

Make a copy of this file, saving it using another name—for example, `galileo-platform-data.ini`—and open this file in the text editor of your preference.

As you can observe, this file is divided into sections such as [Platform Type], [Mrc Params], [MAC address 0], and [MAC address 1]; but what really matters with Intel Galileo boards are the sections [Platform Type] and [MAC address 0]. On each section there is a field called `data.value=` that represents the place you will need to modify the platform data.

In the section [Platform Type,] there is the following comment:

```
# ClantonPeak 2, KipsBay 3, CrossHill 4, ClantonHill 5, KipsBay-fabD 6, GalileoGen2 8
```

If your board is Intel Galileo Gen 2, the `data.value` must receive the value **8**, and although the Intel Galileo is not mentioned, the value must be **6** and must be considered as KipsBay-fabD.

For example, if your board is Intel Galileo Gen 2, the [Platform Type] must be changed in this way:

```
[Platform Type]
id=1
desc=PlatformID
data.type=hex.uint16
# ClantonPeak 2, KipsBay 3, CrossHill 4, ClantonHill 5, KipsBay-fabD 6, GalileoGen2 8
data.value=8
```

The other section that you need to modify is the [MAC address 0], again changing the `data.value` field. For example, suppose your Ethernet MAC address white tag says MAC:984FEE014C6B; then this section must be changed to the following:

```
[MAC address 0]
id=3
desc=1st MAC
data.type=hex.string
data.value=984FEE014C6B
```

Now you need to generate the binary file patching the platform data. First, take a quick look at the option offered by the `platform-data-patch.py` script:

```
mcramon@ubuntu:~/ $ ./platform-data-patch.py --help
Usage: platform-data-patch.py
```

Options:

```
--version          show program's version number and exit
-h, --help         show this help message and exit
-i ORIGINAL_IMAGE, --original-image=ORIGINAL_IMAGE
                  input flash image [default: Flash-missingPDAT.bin]
-p INPUT_FILE, --platform-config=INPUT_FILE
                  configuration (INI) file [default: platform-data.ini]
```

```
-n MODIFIED_IMAGE, --name=MODIFIED_IMAGE
    output flash image [default: Flash+PlatformData.bin]
-u, --undefined-order
    By default, items are put in the same order as they
    come in the config file. However ordering requires
    python 2.7 or above.
```

The script is very simple: the option `-i` indicates that the input capsule file, `-p`, is the platform-data file; `-n` is the name of your **output file**; and `-u` must be used *only* if your version of **Python is lower than 2.7**. So, before using this script, check which Python version is installed on your computer by typing `python --version` on your console to determine if the `-u` option must be used or not. If you have a recent version of Python, you just need to run the script.

In the next example, the output file was named `cool_binary.bin` and the input files were the ones created as examples in this chapter.

```
./platform-data-patch.py -i ../../sysimage/sysimage.CP-8M-debug/Flash-
missingPDAT.bin -p galileo-platform-data.ini -n cool_binary.bin
```

If the script ran smoothly, you should have the output file `cool_binary.bin` in the same directory. Next it is time to flash your image using the SPI flash programmer described in the next section.

If you make mistakes in the **[Platform Type]**, (for example, suppose you specify that the `data.value` equals 6, which means Intel Galileo, but flash an Intel Galileo Gen 2 board), during the boot, the firmware will recognize the incompatibility and will ask you to select the board type, manually displaying a menu that might be seen using the Linux terminal shell in your board.

```
Type '0' for 'ClantonPeakSVP' [PID 2]
Type '1' for 'KipsBay' [PID 3]
Type '2' for 'CrossHill' [PID 4]
Type '3' for 'ClantonHill' [PID 5]
Type '4' for 'Galileo' [PID 6]
Type '5' for 'GalileoGen2' [PID 8]
```

So, if you see this menu, the platform file on your board was not patched, or it was patched with wrong data.

## Flashing Using an SPI Flash Programmer

It is recommended to use the flash programmer called DediProg SF100, which you can order from <http://www.dediprogram.com/pd/spi-flash-solution/sf100>.

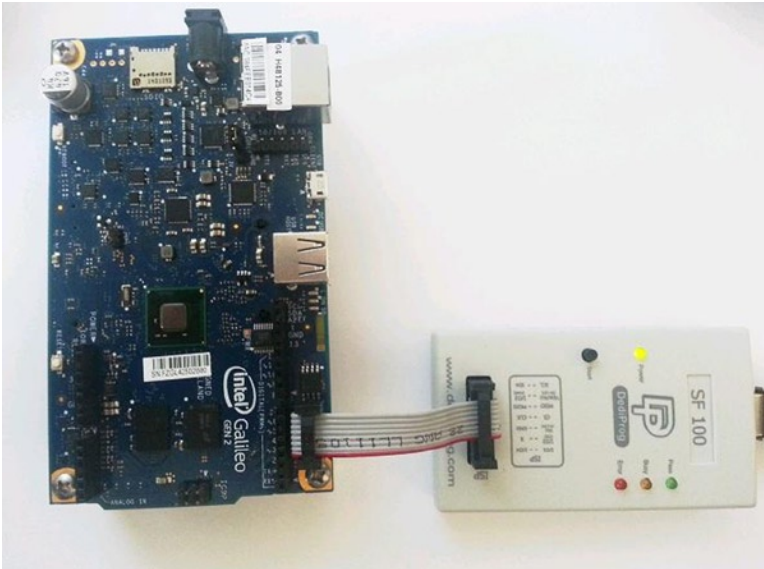
Officially, the DediProg SF100 only works on Windows, but the open source community has a program called *flashrom* that supports DediProg SF 100 on Linux and Mac OSX as well.

This section will focus on DediProg for Windows, but if you are a Linux or Mac OSX developer, you can use **flashrom** with DediProg by downloading from <http://www.flashrom.org/Flashrom> and using the following command line:

```
flashrom -p dediprogram -r biosimage.rom
```

The procedure to use the DediProg SF100 using the GUI interface is as follows:

1. Connect the DediProg SF100 to Intel Galileo by using the SPI programmer terminal, as shown in Figure 2-15, but make sure that Intel Galileo is not connected to any power supply and that the DediProg SF100 is connected to your computer via a USB cable. There is no power supply involved in this process, and the DediProg SF100 uses energy from your USB port. Both the board, Intel Galileo, and Intel Galileo Gen 2 offer the SPI flash port, basically in the same position.



**Figure 2-15.** DediProg SF100 connected to Intel Galileo Gen 2

2. After running the installer that comes with DediProg SF100, run the program **DediProg Engineering**. The first thing that this program will ask about is the SPI flash memory present in the board. If you take a quick look in the Intel Galileo schematics (<https://communities.intel.com/docs/DOC-21822>), you will notice that Intel Galileo and Intel Galileo Gen 2 uses the same type of memory, **W25Q64FV**, as shown in Figure 2-16. Just select the write memory and click the OK button.

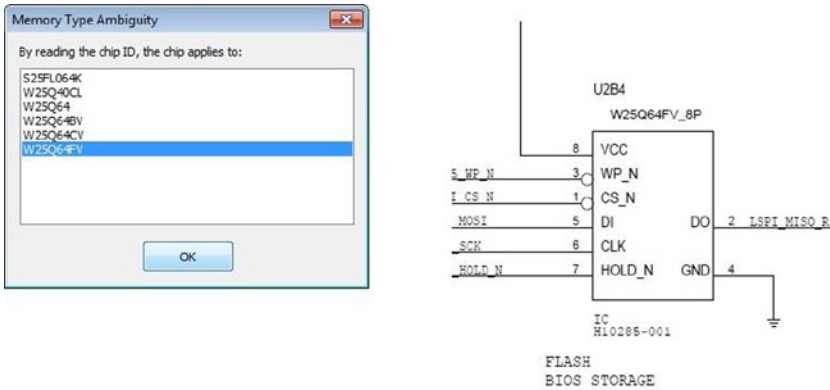


Figure 2-16. Selecting the right SPI flash memory and Intel Galileo schematics

3. Click **Configuration** and change the **Vcc** option to **Manual**. **Select Vcc and 3.5V**; this will save a lot of problems due to flash error. Click the OK button, as shown in Figure 2-17.

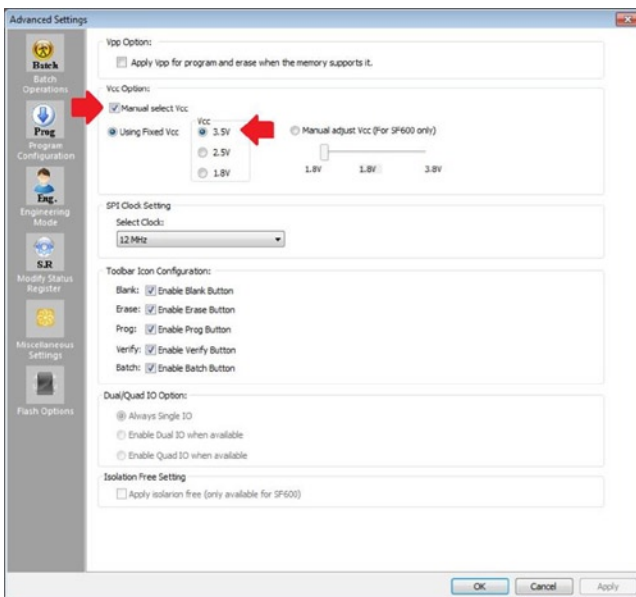
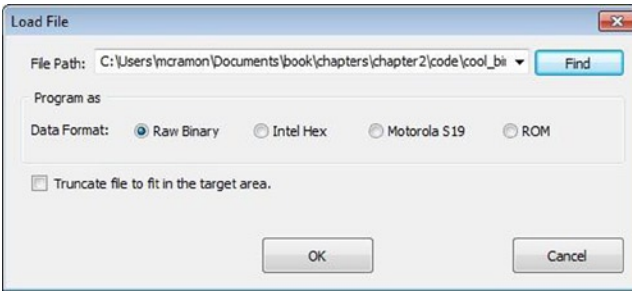


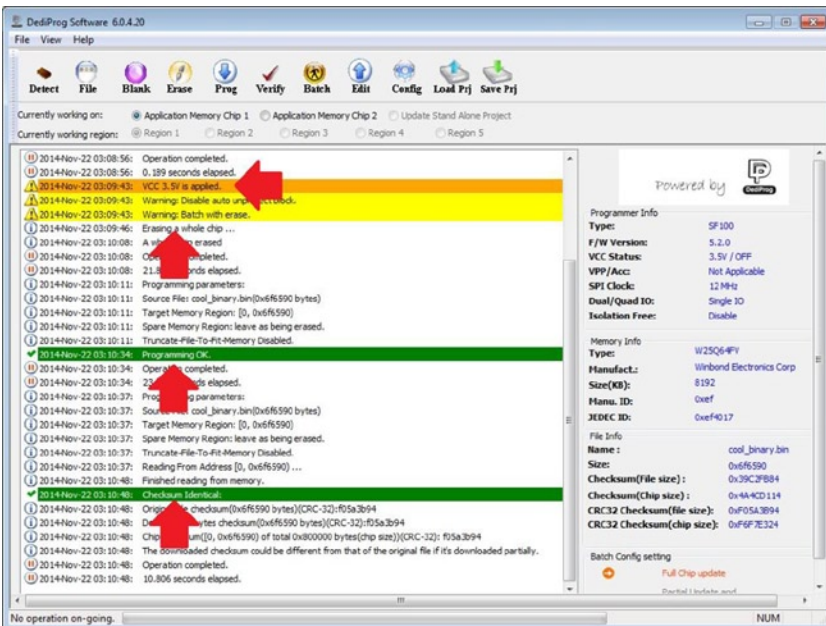
Figure 2-17. Configuring the DediProg SF100 to Vcc 3.5V

4. Click the **File** option and select the binary with the platform you have created, as shown in Figure 2-18. Make sure that the **Raw Binary** option is selected and then click the OK button.



**Figure 2-18.** Selecting the binary file to program

- Now it is time to program. Click the Erase option to erase the SPI flash memory. Then click **Prog** to program the SPI flash memory. Finally, click **Verify** to make sure that your binary was written correctly in the memory. Figure 2-19 shows the process of each step. If the verification fails, repeat this step until you have the SPI flash memory properly programmed. This easily happens if you did not select the 3.5V mentioned in step 3.



**Figure 2-19.** Erasing, programming, and verifying steps in DediProg SF100

## What to Do If Intel Galileo Bricks

There are some situations where your Intel Galileo may be bricked:

- You lost power during the flash.
- The SPI programmer flashed some part of the memory incorrectly, corrupting the SPI, and you could not detect the error because you did not verify.
- You made a mistake in the `layout.conf` file.
- You patched the binaries, declaring a wrong board model in the platform data. For example, you have an Intel Galileo and you incorrectly modified the platform data to Intel Galileo Gen 2.

There are two procedures that might help solve this situation after you fixed and verified, if you did not make any mistakes with the software you created:

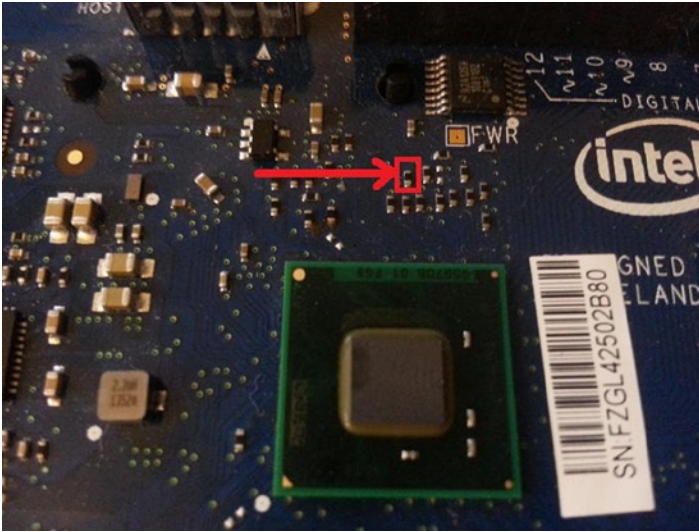
1. Use an SPI flash programmer to reprogram the SPI flash memory. This procedure was mentioned in the “*Flashing Using an SPI Flash Programmer*” section of this chapter.
2. Use `FVMAIN.fv`, created with SPI flash tool from the “*Using the SPI Tool*” section of this chapter.

The procedure to use in the second case is as follows:

1. Power-off Intel Galileo, removing the power supply.
2. Copy `FVMAIN.fv` to a USB pen drive.
3. Keep the serial debug cable (FTDI or serial audio) jack connected and open on a serial software terminal of your preference. Read the “*Preparing Your Own Cables*” section in Chapter 1 if you do not know how to do that.



4. Connect the USB pen drive in the USB OTG port on your Intel Galileo Gen 2. If you are using Intel Galileo, you will need an adaptor like the one shown in Figure 5-20 of Chapter 5.



**Figure 2-20.** Resistor to ground to enter the recovery mode

5. Ground the R2B16 resistor, as shown in Figure 2-20.
6. Connect the power supply to Intel Galileo.
7. In the serial shell, a list of platforms will be shown; choose the Galileo model.
8. Remove the resistor from ground.
9. In the serial shell, select the system recovery option. The system recovery will take around 6 minutes to complete.

These are the two methods that you can try. I wish you sincere good luck with them.

## Summary

In this chapter, you received an overview of how the Yocto build system works and how to generate SD card and SPI releases for Intel Galileo boards, as well as how to generate the toolchain and IPK packages. You could also test the cross-compilers present in the toolchain, creating a simple native program, and then run it on Intel Galileo to test it.

The chapter also explained the differences between capsule and binary files with platform data, how to build firmware on EDKII repositories, and how to recover your board if bad firmware was flashed.