Embedded Systems Programming
OS Linux - Build systems

Iwona Kochańska

Gdansk University of Technology
Elements of embedded Linux

- **Toolchain**: consists of the compiler and other tools needed to create code for target device.
- **Bootloader**: necessary to initialize the board and to load and boot the Linux kernel.
- **Kernel**: heart of the system, managing system resources and interfacing with hardware.
- **Root filesystem**: contains the libraries and programs that are run once the kernel has completed its initialization.
Build System

- Building:
  - a toolchain,
  - a bootloader,
  - a kernel,
  - a root filesystem

and combine them into a basic embedded Linux system

- A lot of steps! “Roll your own” (RYO) process
  - Complete control of the software
  - System can be tailored to do anything
  - To reduce the memory footprint to the smallest possible
In the vast majority of situations, building manually is a waste of time and produces inferior, unmaintainable systems.

The idea of a **build system** is to automate all the steps.

System should be able to build, from upstream source code, some or all of the:

- toolchain
- bootloader
- kernel
- root filesystem
Build system has to be able to:

- Download a source from upstream, either directly from the source code control system or as an archive, and cache it locally
- Apply patches to enable cross compilation, fix architecture-dependent bugs, apply local configuration policies, and so on
- Build the various components
- Create a staging area and assemble a root filesystem
- Create image files in various formats ready to be loaded onto the target
Build system

Other useful functionalities:

▶ Add your own packages containing, for example, applications or kernel changes
▶ Select various root filesystem profiles: large or small, with and without graphics or other features
▶ Create a standalone SDK that you can distribute to other developers so that they don’t have to install the complete build system
▶ Track which open source licenses are used by the various packages you have selected
▶ Allow you to create updates for in-field updating Have a user-friendly user interface
Build system

- Build system **encapsulate** the components of a system into **packages**, some for the **host** and some for the **target**.

- Each package is defined by a set of rules to:
  - get the source,
  - build it,
  - install the results in the correct location.

- There are dependencies between the packages and a build mechanism to resolve the dependencies and build the set of packages required.
Build systems

Open source build systems:

- **Buildroot**: An easy-to-use system using GNU make and Kconfig (http://buildroot.org)
  - the primary aim of building root filesystem images (hence the name)
  - it can build bootloader and kernel images as well
  - easy to install and configure, and generates target images quickly.

- **EmbToolkit**: A simple system for generating root filesystems; the only one at the time of writing that supports LLVM/Clang out of the box (https://www.embtoolkit.org)

- **OpenEmbedded**: A powerful system which is also a core component of the Yocto Project and others (http://openembedded.org)

- **OpenWrt**: A build tool oriented towards building firmware for wireless routers (https://openwrt.org)
Open source build systems:

- **PTXdist**: An open source build system sponsored by Pengutronix (http://www.pengutronix.de/software/ptxdist/index_en.html)
- **Tizen**: A comprehensive system, with emphasis on mobile, media, and in-vehicle devices (https://www.tizen.org)
- **The Yocto Project**: This extends the OpenEmbedded core with configuration, layers, tools, and documentation: probably the most popular system (http://www.yoctoproject.org)
  
  - can build complex embedded devices.
  - every component is generated as a package in RPM, .dpkg or .ipk format
  - the packages are combined together to make the filesystem image
  - creating own custom Linux distribution.
Package formats

- **rpm** (Red Hat Package Manager) - used in Red Hat, Suse, Fedora, and other distributions based on them.
- **deb** - (Debian package manager) - used in Debian-derived distributions, including Ubuntu and Mint
- **ipk** (Itsy PacKage) - light-weight format specific to embedded devices, based on deb.

Ability to include a package manager on the device is one of the big differentiators between build systems.
The Buildroot project website: http://buildroot.org
Documentation: http://buildroot.org/docs.html

Current versions of Buildroot are capable of building:
- a toolchain,
- a bootloader (U-Boot, Barebox, GRUB2, or Gummiboot),
- a kernel,
- a root filesystem.
Buildroot

- Uses **GNU make** as the principal build tool.
- Background:
  - One of the first build systems
  - Began as part of the uClinux and uClibc projects as a way of generating a small root filesystem for testing.
  - Became a separate project in late 2001 and continued to evolve through to 2006
  - Since 2009 - developing rapidly (Peter Korsgaard)
The Buildroot developers produce stable releases four times a year, in February, May, August, and November.

They are marked by git tags of the form `<year>.02`, `<year>.05`, `<year>.08`, and `<year>.11`.

Stable releases are seldom updated after release.
Buildroot - installing

- Cloning the repository or downloading an archive
  
  $ git clone git://git.buildroot.net/buildroot
  $ cd buildroot
  $ git checkout 2015.08.1

Buildroot - configuring

Buildroot uses the **Kconfig** and **Kbuild** mechanisms as the kernel

- configure directly using **make menuconfig**
- choose one of the 90 or so configurations for various development boards and the QEMU emulator (configs/)
- **make help** - lists all the targets including the default configurations.
- buildroot will make optimum use of CPUs (no need to give -j parameter for make)

Example: target - ARM for QEMU

```bash
$ cd buildroot
$ make qemu_arm_versatile_defconfig
$ make
```
Buildroot - building

Results: two new directories

- **dl/**: contains archives of the upstream projects that Buildroot has built
- **output/**: This contains all the intermediate and final compiled resources
  
  - **build/**: build directory for each component.
  - **host/**: contains various tools required by Buildroot that run on the host, including the executables of the toolchain (in output/host/usr/bin)
  - **images/**: results of the build (a bootloader, a kernel, one or more root filesystem images)
  - **staging/**: symbolic link to the sysroot of the toolchain
  - **target/**: staging area for the root directory. Cannot be used as a root filesystem, as it stands, because the file ownership and permissions are not set correctly. Buildroot uses a device table, as described in the previous chapter, to set ownership and permissions when the filesystem image is created.
Sample configurations have a corresponding entry in the directory boards/

Example: start QEMU with ARM compiled target:

```bash
$ qemu-system-arm -M vexpress-a9 -m 256 \ 
-kernel output/images/zImage \ 
-dtb output/images/vexpress-v2p-ca9.dtb \ 
-drive file=output/images/rootfs.ext2,if=sd \ 
-append "console=ttyAMA0,115200 root=/dev/mmcblk0" \ 
-serial stdio -net nic,model=lan9118 -net user
```

login as root, no password.

To close QEMU, type **poweroff** at the root prompt.
Yocto Project

The Yocto Project can build:

- toolchains,
- bootloaders
- kernels
- root filesystems
- entire Linux distribution with binary packages that can be installed at runtime.
Yocto Project

The Yocto Project is primarily a group of recipes
  ► similar to Buildroot packages
  ► written using a combination of Python and shell script
  ► task scheduler called BitBake that produces whatever you have configured, from the recipes.

Online documentation at https://www.yoctoproject.org/.
Yocto Project - background (OpenEmbedded)

- Name: SI prefix for 10-24
- Roots of the Yocto Project - OpenEmbedded, http://openembedded.org/
- OpenEmbedded grew out of a number of projects to port Linux to various hand-held computers, including the Sharp Zaurus and Compaq iPaq.
- OpenEmbedded came to life in 2003 as the build system for those hand-held computers but quickly expanded to encompass other embedded boards.
- It was developed and continues to be developed by an enthusiastic community of programmers.
Compact .ipk format, which could then be combined in various ways to create a target system and be installed on the target at runtime.

Creating recipes for each piece of software and using BitBake as the task scheduler.

Supplying BitBake with metadata
2005 - Richard Purdie (OpenedHand) created **Poky**

- a fork of OpenEmbedded
- more conservative choice of packages
- stable releases

OpenEmbedded and Poky continued to run alongside each other, sharing updates and keeping the architectures more or less in step.

Intel brought out OpenedHand in 2008 and they transferred Poky Linux to the Linux Foundation in 2010 when they formed the **Yocto Project**

Since 2010 - common components of OpenEmbedded and Poky combined into a separate project known as **OpenEmbedded core** (oe-core)
Yocto Project - components

- **Poky**: The reference distribution
- **oe-core**: The core metadata, which is shared with OpenEmbedded
- **BitBake**: The task scheduler, which is shared with OpenEmbedded and other projects
- **Documentation**: User manuals and developer’s guides for each component
- **Hob**: A graphical user interface to OpenEmbedded and BitBake
- **Toaster**: A web-based interface to OpenEmbedded and BitBake
- **ADT Eclipse**: A plug-in for Eclipse that makes it easier to build projects using the Yocto Project SDK
The Yocto Project provides a stable base which can be used as it is or which can be extended using **meta layers**.

Many SoC vendors provide board support packages (BSP) for their devices in this way.

Meta layers can also be used to create extended, or just different, build systems.

- some are open source (i.e. the Angstrom Project)
- others are commercial (i.e. MontaVista Carrier Grade Edition, Mentor Embedded Linux, and Wind River Linux)
Yocto Project - stable releases and support

- release of the Yocto Project every six months (in April and October)
- Four most recent releases:

<table>
<thead>
<tr>
<th>Code name</th>
<th>Release date</th>
<th>Yocto version</th>
<th>Poky version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fido</td>
<td>April 2015</td>
<td>1.8</td>
<td>13</td>
</tr>
<tr>
<td>Dizzy</td>
<td>October 2014</td>
<td>1.7</td>
<td>12</td>
</tr>
<tr>
<td>Daisy</td>
<td>April 2014</td>
<td>1.6</td>
<td>11</td>
</tr>
<tr>
<td>Dora</td>
<td>October 2013</td>
<td>1.5</td>
<td>10</td>
</tr>
</tbody>
</table>
Profesjonalna stacja robocza programisty powinna spełniać następujące wymagania:

- system wieloprocesorowy symetryczny (*symmetric multiprocessing, SMP*)
- co najmniej 8 GB pamięci, szybki dysk twardy
- szybkie łącze internetowe

Wymagane oprogramowanie:

- OS Linux (jako system natywny lub wirtualny)
- Tar w wersji 1.24 lub wyższej
- Python w wersji 2.7.3 lub wyższej (ale nie Python 3)

Instalacja wymaganych pakietów:

```
$ sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib build-essential chrpath socat libSDL1.2-dev xterm make xsltproc docbook-utils fop dblatex xmlto autoconf automake libtool libglib2.0-dev python-gtk2 bsdmainutils screen
```
Yocto Project - configuring

- To get a copy of the Yocto Project:
  
  ```bash
  $ git clone -b fido git://git.yoctoproject.org/poky.git
  ```

- Sourcing a script to set up the environment:
  
  ```bash
  $ cd poky
  $ source oe-init-build-env
  ```

  That creates a working directory for you named build and makes it the current directory.
  
  - All of the configuration, intermediate, and deployable files will be put in this directory.
  - You must source this script each time you want to work on this project.

- Choose a different working directory:
  
  ```bash
  $ source oe-init-build-env build-qemuarm
  ```
Yocto Project - configuring

Initially, the build directory contains only one subdirectory named `conf`, which contains the configuration files for this project:

- `local.conf`: a specification of the device to be build and the build environment
- `bblayers.conf`: a list of the directories that contain the layers you are going to use.
- `templateconf.cfg`: the name of a directory which contains various conf files. By default, it points to `meta-yocto/conf`.

Set the `MACHINE` variable in `local.conf`.

Examples:

- for QEMU emulator:
  
  `MACHINE ?= "qemuarm"

- for Quark processor (Intel Galileo):
  
  `MACHINE ?= "quark"`
Yocto Project - building

Run **Bitbake** with name of root filesystem image:

- **core-image-minimal**: a small console-based system which is useful for tests and as the basis for custom images.
- **core-image-minimal-initramfs**: similar to core-image-minimal, but built as a ramdisk.
- **core-image-x11**: a basic image with support for graphics through an X11 server and the xterminal terminal app.
- **core-image-sato**: a full graphical system based on Sato, which is a mobile graphical environment built on X11, and GNOME. Includes several apps including a terminal, an editor, and a file manager.

Example - create minimal image:

```
$ bitbake core-image-minimal
```
Yocto Project - building

Built directories and files:

- **build/downloads** - all the source downloaded for the build,
- **build/tmp** - most of the build artifacts
  - **work**: the build directory and the staging area for all components, including the root filesystem
  - **deploy**: the final binaries to be deployed on the target:
    - **deploy/images/[machine name]**: the bootloader, the kernel, and the root filesystem images ready to be run on the target
    - **deploy/rpm**: the RPM packages that went to make up the images
    - **deploy/licenses**: the license files extracted from each package
Yocto Project - user’s point of view
The metadata for the Yocto Project is structured into layers, each with a name beginning with `meta`.

**Warstwa (layer)** - grupa medanych definiujących określoną funkcjonalność:

The core layers of the Yocto Project:

- `meta`: OpenEmbedded core
- `meta-yocto`: metadata specific to the Yocto Project, including the Poky distribution
- `meta-yocto-bsps`: board support packages for the reference machines that the Yocto Project supports

The list of layers in which BitBake searches for recipes is stored in:

`<your build directory>/conf/bblayers.conf`

and, by default, includes all three layers.
### Bitbake Layering

<table>
<thead>
<tr>
<th>Additional Layers</th>
<th>Default Layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom Layer</td>
<td>Yocto Layer</td>
</tr>
<tr>
<td>(meta-myproject)</td>
<td>./meta-yocto</td>
</tr>
<tr>
<td>Software Layer</td>
<td></td>
</tr>
<tr>
<td>(meta-baryon)</td>
<td></td>
</tr>
<tr>
<td>Kernel Enablement Layer</td>
<td></td>
</tr>
<tr>
<td>(experimental/meta-x32)</td>
<td></td>
</tr>
<tr>
<td>Hardware Enablement Layer</td>
<td></td>
</tr>
<tr>
<td>(meta-intel/meta-fsl-ppc)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OE-core</td>
</tr>
<tr>
<td></td>
<td>./meta</td>
</tr>
</tbody>
</table>
Additional layers are available from:

- SoC manufacturers,
- the Yocto Project itself,
- a wide range of people wishing to add value to the Yocto Project and OpenEmbedded.

Useful list of layers:

http://layers.openembedded.org
Yocto Project - layers

Example: bblayers.conf for Intel Galileo

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

BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= " \
/opt/yoctoIG/bsp/meta-clanton_v1.2.1/meta \
/opt/yoctoIG/bsp/meta-clanton_v1.2.1/meta-yocto \
/opt/yoctoIG/bsp/meta-clanton_v1.2.1/meta-intel-iot-devkit \
/opt/yoctoIG/bsp/meta-clanton_v1.2.1/meta-intel-iot-middleware \
/opt/yoctoIG/bsp/meta-clanton_v1.2.1/meta-intel-quark \
/opt/yoctoIG/bsp/meta-clanton_v1.2.1/meta-intel-galileo \
/opt/yoctoIG/bsp/meta-clanton_v1.2.1/meta-netcontiki \
/opt/yoctoIG/bsp/meta-clanton_v1.2.1/meta-openembedded/meta-networking \
"

BBLAYERS_NON_REMOVABLE ?= " \
/opt/yoctoIG/bsp/meta-clanton_v1.2.1/meta \
/opt/yoctoIG/bsp/meta-clanton_v1.2.1/meta-yocto \
"
Yocto Project - Bitbake and recipes

BitBake processes metadata of several different types, which include:

- **recipes**: files ending in `.bb`. These contain information about building a unit of software, including how to get a copy of the source code, the dependencies on other components, and how to build and install it.

- **append**: files ending in `.bbappend`. These allow some details of a recipe to be overridden or extended. A.bbappend file simply appends its instructions to the end of a recipe (.bb) file of the same root name.

- **include**: files ending in `.inc`. These contain information that is common to several recipes, allowing information to be shared among them. The files may be included using the include or require keywords. The difference is that require produces an error if the file does not exist, whereas include does not.
Yocto Project - Bitbake and recipes

- **classes**: files ending in `.bbclass`. These contain common build information, for example how to build a kernel or how to build an autotools project. The classes are inherited and extended in recipes and other classes using the inherit key word. The class `classes/base.bbclass` is implicitly inherited in every recipe.

- **configuration**: files ending in `.conf`. They define various configuration variables that govern the project's build process.
A recipe is a collection of tasks written in a combination of Python and shell code.

<table>
<thead>
<tr>
<th>ZADANIE</th>
<th>OPIS</th>
<th>FUNKCJA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>download sources</td>
<td>do_fetch()</td>
</tr>
<tr>
<td>Unpack</td>
<td>unpack sources</td>
<td>do_unpack()</td>
</tr>
<tr>
<td>Patch</td>
<td>adding patches</td>
<td>do_patch()</td>
</tr>
<tr>
<td>Configure</td>
<td>source tree configuration</td>
<td>do_configure()</td>
</tr>
<tr>
<td>Compile</td>
<td>source tree compilation</td>
<td>do_compile()</td>
</tr>
<tr>
<td>Stage</td>
<td>installation in stage space</td>
<td>do_stage()</td>
</tr>
<tr>
<td>Install</td>
<td>installation</td>
<td>do_install()</td>
</tr>
<tr>
<td>Package</td>
<td>building a package</td>
<td>do_package()</td>
</tr>
</tbody>
</table>
Yocto Project - Bitbake and recipes

Source: P. Raghavan, Amol Lad, Sriram Neelakandan, “Embedded Linux system design and development”
Yocto Project - Bitbake and recipes

1. Analiza przepisów (recipes) i plików konfiguracyjnych aby ustalić co i jak ma być zbudowane
2. Pobranie z sieci kodu źródłowego
3. Budowa obrazu systemu
Bitbake - making images

Process of Making Images

- BitBake Recipes
- Binary Packages
- Task Graph
- Flash Image

źródło: www.denx.de/wiki/
Yocto Project - architecture (openembedded)
BitBake - Hob

Select a machine

Your selection is the profile of the target machine for which you are building the image.

quark

Layers
Add support for machines, software, etc.

Select an image recipe

Image recipes are a starting point for the type of image you want. You can build them as they are or edit them to suit your needs.

core-image-minimal

Advanced configuration
Select image types, package formats, etc.

A small image just capable of allowing a device to boot.
# BitBake - Hob

## Step 1 of 2: Edit recipes

<table>
<thead>
<tr>
<th>Recipe name</th>
<th>Group</th>
<th>Brought in by (+others)</th>
<th>Included</th>
</tr>
</thead>
<tbody>
<tr>
<td>aci</td>
<td>libs</td>
<td>systemd</td>
<td>✓</td>
</tr>
<tr>
<td>attr</td>
<td>libs</td>
<td>libcap (+1)</td>
<td>✓</td>
</tr>
<tr>
<td>base-files</td>
<td>base</td>
<td>dbus (+3)</td>
<td>✓</td>
</tr>
<tr>
<td>base-passwd</td>
<td>base</td>
<td>dbus (+4)</td>
<td>✓</td>
</tr>
<tr>
<td>bash</td>
<td>base/shell</td>
<td>acl (+1)</td>
<td>✓</td>
</tr>
<tr>
<td>bc</td>
<td>base</td>
<td>openssl</td>
<td>✓</td>
</tr>
<tr>
<td>binutils-cross-i586</td>
<td>devel</td>
<td>gcc-cross-i586 (+2)</td>
<td>✓</td>
</tr>
<tr>
<td>busybox</td>
<td>base</td>
<td>packagegroup-core-boot</td>
<td>✓</td>
</tr>
<tr>
<td>bzip2</td>
<td>console/utils</td>
<td>python</td>
<td>✓</td>
</tr>
<tr>
<td>coreutils</td>
<td>base</td>
<td>attr (+1)</td>
<td>✓</td>
</tr>
<tr>
<td>cryptodev-linux</td>
<td>base</td>
<td>openssl</td>
<td>✓</td>
</tr>
<tr>
<td>db</td>
<td>libs</td>
<td>python (+1)</td>
<td>✓</td>
</tr>
<tr>
<td>dbus</td>
<td>base</td>
<td>glib-2.0 (+3)</td>
<td>✓</td>
</tr>
<tr>
<td>dbus-glib</td>
<td>base</td>
<td>python-dbus</td>
<td>✓</td>
</tr>
<tr>
<td>depmodwrapper-cross</td>
<td>base</td>
<td>core-image-minimal (+1)</td>
<td>✓</td>
</tr>
<tr>
<td>diffutils</td>
<td>base</td>
<td>gdbm</td>
<td>✓</td>
</tr>
<tr>
<td>e2fsprogs</td>
<td>base</td>
<td>libsm (+1)</td>
<td>✓</td>
</tr>
<tr>
<td>expat</td>
<td>libs</td>
<td>gettext (+3)</td>
<td>✓</td>
</tr>
<tr>
<td>flex</td>
<td>devel</td>
<td>bc</td>
<td>✓</td>
</tr>
<tr>
<td>gawk</td>
<td>console/utils</td>
<td>libcheck</td>
<td>✓</td>
</tr>
<tr>
<td>gcc-cross-i586</td>
<td>devel</td>
<td>packagegroup-core-boot (+70)</td>
<td>✓</td>
</tr>
<tr>
<td>gcc-cross-initial-i586</td>
<td>devel</td>
<td>libgcc-initial (+2)</td>
<td>✓</td>
</tr>
<tr>
<td>gcc-runtime</td>
<td>devel</td>
<td>packagegroup-core-boot (+67)</td>
<td>✓</td>
</tr>
</tbody>
</table>
BitBake - Hob

Building packages ...

Build Completed: 90%

<table>
<thead>
<tr>
<th>Build configuration</th>
<th>Issues</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>Message</td>
<td></td>
</tr>
<tr>
<td>Resolving any missing task queue dependencies</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Build Configuration:
BB_VERSION  = "1.24.0"
BUILD_SYS   = "x86_64-linux"
NATIVEARCHSTRING  = "Debian-8.2"
TARGET_SYS   = "i586-poky-linux"
MACHINE      = "quark"
DISTRO       = "iot-devkit"
DISTRO_VERSION = "1.5"
TUNE_FEATURES = "m32 i586"
TARGET_FPU   = ""
meta
meta-yocto
meta-intel-iot-devkit
meta-intel-iot-middleware
meta-intel-quark
meta-intel-galileo
meta-netcontiki
meta-networking  = "master:25a1cb0a72d03de49df1b90b5c99ca565f963b9e"

Preparing runqueue
Executing SetScene Tasks
Executing RunQueue Tasks
Tasks Summary: Attempted 1741 tasks of which 1741 didn't need to be rerun and all succeeded.
Building Yocto for Intel Galileo

Intel® Quark™ SoC X1000 Board Support Package (BSP) Build and Software User Guide

Release: 1.2.1

February 2016

https://downloadcenter.intel.com/download/23197/Intel-Quark-BSP
Building Yocto for Intel Galileo

2 Before You Begin

Before you begin:

- You need a host PC running either:
  - Linux*; Intel recommends a 64-bit Linux system
  - Microsoft* Windows* 7, x64
- You need an internet connection to download third party sources.
- The build process may require as much as 30 GB of free disk space.
- To program the board you can use:
  - A serial interface using the UEFI shell or Linux* run-time (see Section 10)
  - A DediProg* SF100 SPI Flash Programmer (or equivalent) and the associated flashing software (see Section 11)
  - An Intel® Galileo IDE (Intel® Galileo Gen 2 board only; refer to Appendix A Related Documents for User Guide details)

Note: Remove all previous versions of the software before installing the current version.

Individual components require very different environments (compiler options and others). To avoid cross-pollution, the commands in each section that follows must be run in a new command line window every time.
Building Yocto for Intel Galileo

If building on a Debian host PC, use the Debian-provided meta package called build-essential that installs a number of compiler tools and libraries. Install the meta package and the other packages listed in the command below before continuing:

```
# sudo apt-get install build-essential gcc-multilib vim-common
```

Install older versions of gcc and g++ (4.9)!
http://askubuntu.com/questions/26498/choose-gcc-and-g-version
4 Building the EDKII Firmware

You need to build the open source EDKII firmware for the Intel® Quark™ SoC. Additional details may be found here:

- [www.tianocore.sourceforge.net](http://www.tianocore.sourceforge.net)

4.1 Dependencies

Linux* build environment dependencies:

- Python 2.6 or 2.7 (Python 3.x not supported)
- GCC and G++ (tested with GCC 4.3 and GCC 4.6)
- subversion client
- uuid-dev
- nasm
- iasl ([https://www.acpica.org/downloads/linux](https://www.acpica.org/downloads/linux))

**Note:** An ACPI15.0 compatible version is required.
Building Yocto for Intel Galileo

3. Optionally, if OpenSSL is required by the build configuration in Section 4.3 or Section 4.4 following, then perform the steps in the CryptoPkg/Library/OpensslLib/Patch-HOWTO.txt file.

4.2.1 Performing pre-build Steps in a Linux/gcc Build Environment

Open a new terminal session and enter the following commands:

# sudo apt-get install build-essential uuid-dev iasl subversion nasm

# tar -xvf Quark_EDKII_*.*.tar.gz

# cd Quark_EDKII*

# ./svn_setup.py

# svn update
4.3 Building all the EDKII Firmware Validated Build Configurations [Linux build environment only]

This section is only supported in Linux Build environments. The buildallconfigs.sh file is used to build all the validated EDKII build configurations. Open a terminal window and cd to the Quark_EDKII* directory created in Section 4.2.1 Performing pre-build Steps in a Linux/gcc Build Environment.

The script has the following options:

```
buildallconfigs.sh [GCC44 | GCC45 | GCC46 | GCC47 | GCC48 | GCC49] [GCC Path]
[PlatformName]
```

GCC4x

GCC flags used for this build. Set to the version of GCC you have installed.
Note: Tested on GCC46. GCC43 is not supported from release 1.2 onwards.

GCC Path

Location where the GCC is installed. GCC Path is not needed for release 1.1 and earlier.

[PlatformName]

Name of the platform package you want to build.

Example usage:

Create a build for an Intel® Quark™ SoC platform based on GCC version 4.6:

For release 1.2 onwards:

```
./buildallconfigs.sh GCC46 /usr/bin/ QuarkPlatform
```
Building Yocto for Intel Galileo

Creating a File System and Building the Kernel Using Yocto Project*

Dependencies:
- git
- diffstat
- texinfo
- gawk
- chrpath
- lzop
- file

oraz: libstd1.2-dev gcc-4.9-multilib patchutils
Building Yocto for Intel Galileo

First, open a new terminal session, extract the Yocto Project layer, and run the setup.sh script to download the external sources required for the Yocto Project build:

```bash
# tar -xvf meta-quark*.tar.gz
# cd meta-quark*
# ./setup.sh
```

*Note:* The setup.sh script takes no parameters.

Next, source the oe-init-build-env command to initialize the Yocto Project build environment. This command takes the build directory name as its parameter:

```bash
# source ./oe-init-build-env yocto_build
```
Building Yocto for Intel Galileo

# bitbake image-full

When the image build completes, the following output files are found in the ./tmp/deploy/images/quark/ directory. Copy these files to the media device. Rename the files according to the following instructions (resulting paths are relative to the media’s root path):

- `image-full-quark-YYYYMMDDhhmmss.rootfs.ext3`
  (Rename this file to `image-full-quark.ext3` because the initramfs will look for the file in that name, unless the configuration is updated).
- `core-image-minimal-initramfs-quark-YYYYMMDDhhmmss.rootfs.cpio.gz`
  (Optional: rename this file to `core-image-minimal-initramfs-quark.cpio.gz`)
- `bzImage--3.14-r0-quark-YYYYMMDDhhmmss.bin`
  (Optional: rename this file to `bzImage.bin`)
- `grub.elf`
- `boot (directory)`