Politecnico di Torino
Academic Year 2017/2018

Master Thesis

Master in Computer Engineering (Software and Digital Systems)

Embedded Linux distro development with the Yocto Project

Supervisor:
Prof. Massimo Violante
Supervisor (Magneti Marelli):
Cristiano Sponza

Candidate:
Alessandro Flaminio
S241548

Turin, October 2018

Embedded Linux distro development with the Yocto Project

Alessandro Flaminio

Supervised by:

Prof. Massimo Violante

Politecnico di Torino

Cristiano Sponza

Magneti Marelli

Abstract

The main purposes of this thesis are to develop an embedded Linux distribution for a custom Arm-based board, and to provide a new simpler development workflow in the context of the Magneti Marelli Connectivity team. All of these objectives were reached by using the Yocto Project set of tools, and as output the TIIC[1] Linux distro and an SDK[2] to be used by the development team were produced. During this thesis a bottom-up approach was employed: starting from a study of the Yocto Project technologies, to the development of a BSP[3] for the Magneti Marelli Step 03 custom board and, finally, to the automation of the compilation of the Marelli Connectivity Framework software modules. All of these outputs are represented by a set of metadata, contained in the built Yocto Project `meta-mm` layers.

[1] **TIIC**: Technology Innovation, Innovation Connectivity
[2] **SDK**: Software Development Kit
[3] **BSP**: Board Support Package

Dedication

I would like to express my gratitude to my family, in particular to my mother and my father. They always supported me during this long path and I truly believe that it is thanks to them that I am achieving this result.

Contents

List of Figures

List of Tables

Chapter 1

Introduction, Motivations and Goals

1.1 Introduction

An **embedded system** is a computing device specifically crafted for a limited set of purposes in a particular context. Embedded systems control many electrical and mechanical devices that we use and, nowadays, they are almost everywhere. We have embedded systems in our pocket, in our homes and, especially, in our **vehicles**.
These systems are extremely peculiar and application-specific by definition, and, for that reason, their development is fundamentally different from general purpose systems. An embedded system is composed by custom-made hardware and specifically developed software; for that reason specific tools are needed for working proficiently on the development process of both.
In the automotive context, embedded systems are employed for many different tasks such as the **ECU** (Engine Control Unit), **infotainment systems**, safety systems, etc.
The last years have been characterised by an emerging demand for **smarter** vehicles, and one of the hottest topics in this context has been the **vehicular communication**. The **V2X** (Vehicle-to-Everything) technology enables cars to communicate to each other, to city infrastructures, to pedestrians, etc. Of course, as vehicular communication is fulfilled via a coupling of hardware and software components, the development of new **V2X embedded platforms** is needed.
The case study of this thesis is based on the **Magneti Marelli Connectivity (V2X) Framework**, a work that aims to implement the V2X vehicular and infrastructure communication. Being that an embedded system development project, developed in a big company like Magneti Marelli, several challenges need to be addressed. By using state of the art technologies, the development, testing and deploying processes could be vastly improved and simplified.

1.2 Magneti Marelli

Magneti Marelli was founded in **1919**, starting from the 1891 company Ercole Marelli, as a company specialised in **ignition magnetos**. In the last 100 years Magneti Marelli expanded its portfolio to **body control systems**, powertrain control systems, **electronic instrument clusters**, automotive lightning systems, suspension systems, motorsport and other products. Magneti Marelli is a big player in the embedded systems industry, developing and producing embedded platforms for several purposes in the automotive industry.
Magneti Marelli has also several R&D (Research & Development) divisions such as the Autonomous Driving and the **Innovation Connectivity** ones. In particular, the Innovation Connectivity team is developing a full solution for enabling V2X communication.

1.3 Motivations

As already cited, this work is focused on the development process of the Magneti Marelli Connectivity Framework. The procedures adopted by the development team, before the introduction of the workflow developed through this work, were mostly **custom-made** and so, for that reason, were difficult to maintain and to extend. In fact, the existing methods require an extensive knowledge of the environment and several **human interactions** are required. The main areas of improvement that can be identified in the "older" development workflow are the following:

- Compiling and deploying the **Linux kernel** must be performed manually.
- The **BSP** (Board Support Package) must be generated manually.
- Generating **complete images** of the embedded operating system, including the Connectivity software modules, is difficult because all the required applications must be integrated separately.
- **Updating** the existing libraries and utilities requires manual intervention.
- Keeping track of the development **toolchains** distributed among different developers is cumbersome.
- Creating a new project release version requires a manual rewriting of several files.
- The developed embedded software is not easily **scalable** to other platforms and architectures.

1.4 Goals

Given the points reported above, the aim of this thesis is to find ways to enhance the development workflow with the following improvements:

- Compilation of the custom embedded Linux kernel without manual interaction.
- Automated generation of the BSP specific to a board.
- Automatic compilation of the software to be deployed on the system.
- Easy update of libraries and utilities used on the embedded device.
- Seamless generation of a unique development toolchain.
- Simplified project versioning management.
- Simple scalability of the whole software to other platform and architectures.

All of those improvements can be achieved by using one of the most versatile, comprehensive and powerful build systems in the embedded Linux field: the **Yocto Project**.
In particular, this work was targeted towards guaranteeing complete support to the custom **Arm**-based Magneti Marelli **Step 03** board but, given the great scalability of the Yocto Project, a porting to the **x86** architecture was also easily performed.
In particular, the following deliverables were produced as result of this work:

- `meta-mm`, `meta-mm-distro`, `meta-mm-connectivity` **metadata**: a set of metadata to be parsed by the tools included in the Yocto Project that contains all the rules, configurations, parameters, etc. needed for fulfilment of all the objectives reported above.
- **eSDK**: the set of all the tools needed by developers in order to write, test and deploy software on a target embedded platform. The eSDK is automatically generated starting from the Yocto Project and its metadata.

Chapter 2

Embedded Linux

Over the last years the Linux kernel has significantly gained popularity as operating system for embedded platforms. In the following chapter are detailed the most relevant features of the Linux kernel in the context of embedded systems.

2.1 Embedded Linux

Since 2000, the Linux kernel and several open-source components related to it are increasingly used in the context of embedded systems. The Linux kernel, when used in conjunction with other software units in embedded systems, is often referred as Embedded Linux. At the time of writing Linux is de facto the standard choice for embedded platforms. Linux have in general some key advantages, some of which are particularly relevant in embedded environments:

- **Open-source**: All the Linux source code is freely available online, which means that all the kernel components are maintained by a large community of developers. **Reusability** is also highly encouraged because the open-source ecosystem already provides all the elements needed for typical requirements, but also for specific uses.

- **High quality**: The components of the Linux kernel are widely-used and, being developed by a large community, are maintained by several developers.

- **Low cost**: Since Linux is free of charge, there are no licenses to be paid when deploying it on an embedded platform.

- **Customisability**: The kernel behaviour and features can be easily customised by directly modifying the open-source code. Any modification to the operating system can be performed in any moment, without having to rely on third-parties.

The main components that characterise an Embedded Linux distribution are: **The Linux kernel**, a `rootfs` (containing utilities, libraries and applications such as **BusyBox**) and a **bootloader**.
In order to develop software for an embedded platform, several development tools are needed: a **cross-compiling toolchain**, build systems to ease the compilation phases of the software such as **CMake** and **autotools** and, eventually, an **IDE** for coding.

2.1.1 The Linux kernel

The **Linux** kernel was created in 1991 by Linus Torvalds, and nowadays is one of the most successful open-source projects ever created. It is a **monolithic** kernel, which means that the whole operating system works in kernel space. The main features of the Linux kernel are the following:

- **Portability**: can be compiled and run on the most popular architectures.
- **Scalability**: it is possible to run it on embedded devices up to supercomputers.
- **Compatibility**: Linux is compliant to the Single **UNIX** Specification and to **POSIX**, thus ensuring source code portability.
- **Security** and **Reliability**: being open-source, every Linux component is reviewed by many expert developers.
- **Modularity**: thanks to its complete configuration utility, the kernel can be customised to include only what it is needed.

Kernel configuration (Kconfig)

In order to minimise the size of the resulting compiled kernel, a complete configuration mechanism is provided by Linux. Starting from the kernel source code, several options can be configured:

- Architecture specifications
- Device drivers
- Filesystem drivers
- Network protocols

The Linux kernel configuration is saved into a `.config` file located in the root directory of the kernel source code. This file contains a set of tuples `key=value` that represents each kernel compilation option. It is strongly advised not to edit

manually configuration files because some options can have several dependencies. Various interfaces can be used for managing `.config` files that also take care of all the dependencies such as `make menuconfig`. All of those interfaces can be invoked by using `make` with the `Makefile` located in the root directory of the Linux source code.
There are several default configuration files for each **CPU architecture**, but new ones can be created with `make savedefconfig`. This command creates a new `defconfig` file containing all the selected options.

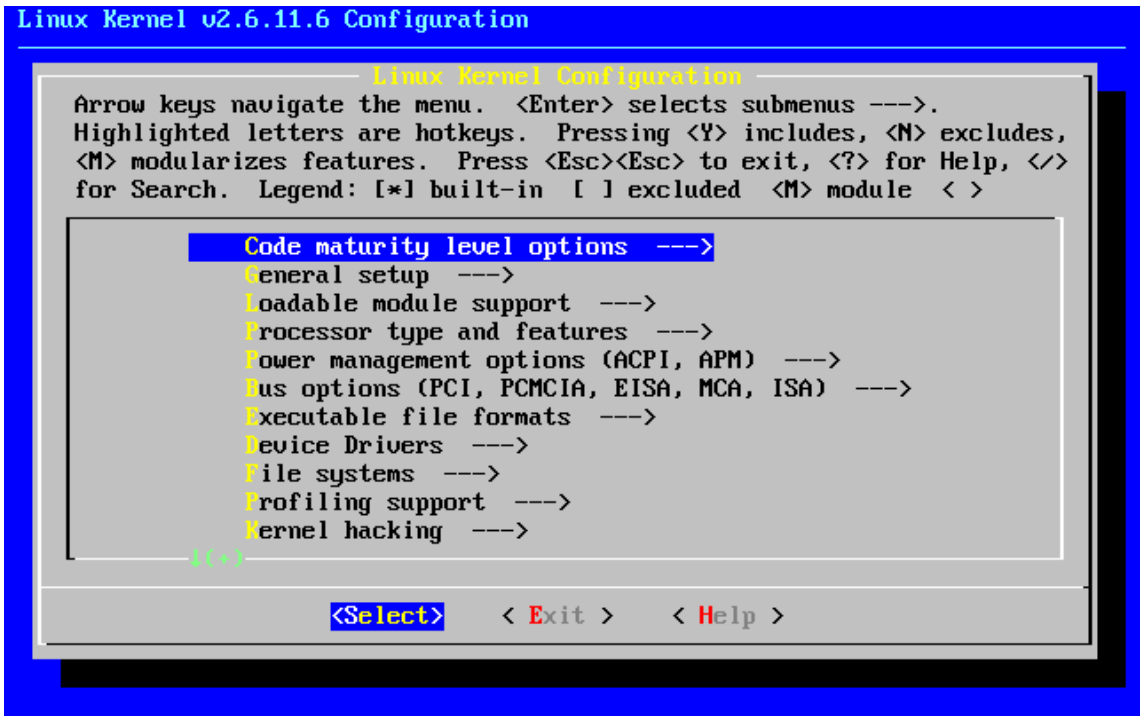


Figure 2.1: The `make menuconfig` graphical configuration utility.

Kernel compilation

After having correctly configured the kernel and having obtained a `.config` file, kernel compilation can be started by simply issuing the `make` command. The build process produces as output a set of files (kernel image and, eventually, the `initramfs` image), depending on the configuration options.

Kernel images

A kernel **image** contains all the objects that have been compiled based on the selected configuration options. The kernel image is a **single file** that is loaded from

the bootloader when starting the system. These are some of the most used types of kernel image:

- **Image**: an uncompressed file that contains all the informations needed to obtain a live working copy of the Linux kernel.
- **zImage**: a compressed version of the **Image**. It is self-extracting when being loaded from the bootloader.
- **uImage**: a file that wraps a generic kernel image with an header used by the **U-Boot** bootloader.
- **fitImage** (Flattened Image Tree): similarly to the **uImage**, it contains U-Boot-specific informations plus other data such as the **DTB** files to be loaded from the U-Boot in order to pass the correct hardware informations to the kernel.

Kernel headers

The Linux kernel headers are a set of header files that represents the **APIs** exposed by the Linux kernel. These files are used in order to correctly compile the kernel modules for a specific version of the Linux APIs.

Kernel modules

Based on the kernel configuration, some features (such as **device drivers**) can be unbundled from the kernel image and included separately in the `rootfs` as **kernel modules** (`.ko` files). In general, installing a driver for a certain peripheral on Linux means compiling a kernel module by using the kernel headers of the operating system. Modules can be loaded and unloaded dynamically after the filesystem is mounted by the kernel, for that reason kernel modules are available only after the initial boot phase.

Device Tree Source & Blob

DTB (Device Tree Blob) files allow a Linux kernel compiled for a specific architecture to boot with different hardware configurations. The Device Tree Blob is produced from a `dtc` (Device Tree Compiler) starting from **DTS** (Device Tree Source) files. Follows an example fragment of a DTS file (taken from [6]) that specifies some hardware informations to the kernel in order to make it recognise a **flash memory**:

```
localbus@e0000000 {
    #address-cells = <2>;
    #size-cells = <1>;
    compatible = "simple-bus";
```

```
        reg = <0xe0000000 0x5000>;
        interrupt-parent = <&mpic>;

        ranges = <0x0 0x0 0xff000000 0x01000000>;   /*16MB Flash*/

        flash@0,0 {
            #address-cells = <1>;
            #size-cells = <1>;
            compatible = "cfi-flash";
            reg = <0x0 0x0 0x1000000>;
            bank-width = <2>;
            device-width = <2>;
            partition@0x0 {
                label = "free space";
                reg = <0x00000000 0x00f80000>;
            };
            partition@0x100000 {
                label = "bootloader";
                reg = <0x00f80000 0x00080000>;
                read-only;
            };
        };
    };
```

DTB files, together with the kernel image, are loaded in RAM from the bootloader (figure 2.2), and the address at which the DTB data is located is passed to the kernel (via the CPU **r2** register) as a parameter. In this way the kernel is able to correctly recognise the specified peripherals.
As said above, DTB files can be included in the kernel **fitImage**, but it is also possible to manually load standalone DTB files via the bootloader.

2.1.2 The `rootfs`

In general, **filesystems** are used in order to organise as a hierarchy files and directories on storage devices. In **UNIX** systems, filesystems are **mounted** in specific directories so that applications can access to filesystems as simple folders in the hierarchy. For that reason, every Linux system must have a **root filesystem** (called `rootfs`) in which other filesystems can be eventually mounted. The `rootfs` is mounted automatically during the boot phase by the kernel at the root of the global hierarchy (`/`), and cannot be mounted manually as others filesystems. The location of the `rootfs` is specified by a parameter passed to the kernel by the bootloader. There are two ways to handle the mounting of the `rootfs` during the boot process [2]:

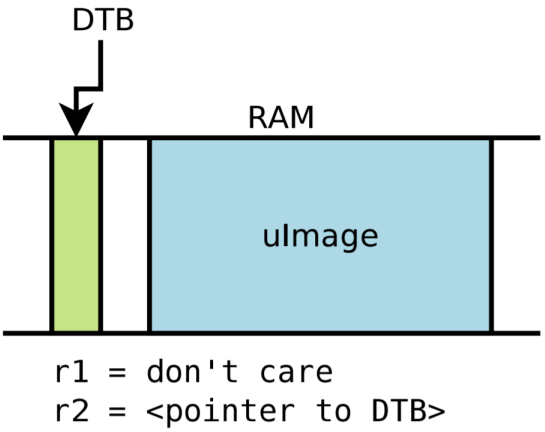


Figure 2.2: The DTB is loaded in RAM with the kernel image.

- **Without a minimal filesystem**: with this technique the `rootfs` is mounted during the kernel initial tasks:

  1. **Bootloader**: loads the kernel (and eventually the DTB) passing to it the `root=` parameter and starts it.
  2. **Kernel**: Prepares devices and kernel components, mounts the `rootfs` based on the `root=` parameter and starts the init manager that completes the system startup.

- **With the `initramfs`**: with this technique the `rootfs` is mounted after mounting a minimal filesystem (called `initramfs`):

  1. **Bootloader**: loads the kernel (and eventually the DTB) and loads the `initramfs` archive (if it is not included in the kernel image).
  2. **Kernel**: Prepares devices and kernel components, mounts the `initramfs` and starts a basic init manager.
  3. `init`: starts some user space commands, loads drivers to access the real `rootfs`, mounts the `rootfs` and starts the complete init manager that finishes the system startup.

Synthetic filesystems

Synthetic (virtual) filesystems are used by Linux to expose statistics and system informations to the user and to user space applications. Synthethic filesystems are mounted exactly like standard filesystems, but the data that they represent are generated by the kernel. There are two main virtual filesystems:

- `/proc`: provides statistics about running processes and implements (via the virtual directory `/proc/sys`) a way for adjusting some kernel parameters. Applications like `ps` require the abstraction provided by `/proc`.
- `/sys`: this virtual filesystem gives to the user space the representation of the buses, devices and drivers as seen by the kernel. This abstraction is needed by applications that require a list of the available hardware.

Libraries

One of the key elements included in the `rootfs` (under the `/lib` directory) are the **dynamic libraries** used by the applications. In particular, the `rootfs` certainly contains all the system libraries, such as the C library (`libc`). Non-basic libraries can be included in the `/usr/lib` directory.

Applications

Basic programs (such as the command **shell**, etc.) are included in the `/bin` folder, basic system binaries are in the `/sbin` folder (such as the init manager, the `mount` command, etc.), while non-basic binaries are put in the `/usr/bin` and `/usr/sbin` folders.

BusyBox

Every Linux system requires several binaries to be able to function properly such as the **init manager**, the **shell** and several other utilities. In standard Linux systems all of those binaries are maintained separately and are integrated into the system with all the features they offer. In the context of embedded systems this is not desirable because every bit of storage can be crucial, and likely not all the features provided by the standard binaries are required. **BusyBox** provides a single binary containing every needed application rewritten into a single project, with all the needed utilities represented by symbolic links to it. When compiling BusyBox, it is also possible to configure the applications in a way to strip certain unneeded features.

2.1.3 Init managers

In Linux systems init managers have the purpose to startup services during the boot process. Most services are in the form of **daemons**, and there could be several dependencies between one service and another. The startup process is organised in multiple steps, identified by **runlevels**. Essential services are put on **lower** runlevels, so that non-basic services can be started afterwards.
The two main init managers employed in Linux systems are the following:

- **SysVinit**: this is the most used init manager in Linux, derived from the **UNIX System V** init style.
- `systemd`: created to be the successor to SysVinit, as it was designed to overcome the shortcomings of it.

2.1.4 Device managers

A device manager is a component that manages devices, making them virtually available in the `/dev` directory. The de facto standard device manager is `udev` that, when using `systemd` as init manager, is incorporated in it. After the **device drivers** recognise the peripherals, `udev` is notified, which then "mounts" the devices under the `/dev` directory. There are two main types of devices that are treated differently by `udev`:

- **Character devices**: also known as **raw devices**, are peripherals to which data is transferred one character at a time. Typical examples of character devices are serial connections, keyboards, etc.
- **Block devices**: devices that support the transfer of (buffered) blocks of data. Typical devices are hard disks, pen drives, etc.

2.2 Flash memory support

One of the most used storage options in embedded devices is flash memory. This kind of memories has specific requirements such as **wear leveling** for preserving endurance over time. The Linux kernel was extended for completely supporting flash memories and several file systems such as **UBIFS**, YAFFS, JFFS2, etc. were specifically created. In particular, one of the most used architecture of subsystems used in flash-based devices is the following [14] (figure 2.3):

- **MTD subsystem**: provides an interface to access **raw** flash chips.
- **UBI subsystem**: manages the **wear-leveling** and the UBI volumes management of the flash device.
- **UBIFS file system**: placed on top of UBI volumes.

2.2.1 The MTD subsystem

The MTD (Memory Technology Device) subsystem provides a device file in Linux for accessing **raw** flash memory. With this abstraction layer, flash devices are represented in Linux as `/dev/mtd` MTD devices. MTD is used with devices that do not provide an MTL (Memory Translation Layer).

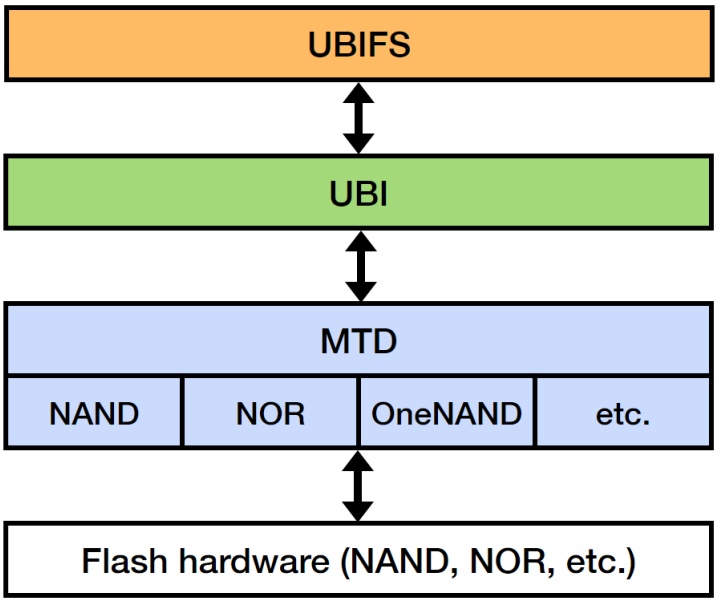


Figure 2.3: The UBI/UBIFS stack.

2.2.2 UBI and the UBI File-System

UBI (Unsorted Block Images) is a layer placed on top of MTD that supervises the bad block management of the flash device (tracking of bad blocks, wear leveling) and provides the notion of UBI volumes that can be dynamically created, removed and re-sized.
UBIFS (Unsorted Block Image File System) is the filesystem that lies on top of the UBI subsystem and is characterised by the following main features:

- **Improved speed**: UBIFS presents improved speeds (with respect to JFFS2) in several operations like I/O, mounting and accessing large files.
- **Power-cuts tolerance**: since UBIFS is a **journaled** file system (it keeps a log of all the operations that performs on the media), it can easily recover from unclean shutdowns.
- **Integrity**: UBIFS creates checksums for everything written on the media, so that data corruptions are not unnoticed.

2.3 *Das U-Boot* bootloader

U-Boot is an open source bootloader primarily used in embedded devices. U-Boot is both a **first-stage** and a **second-stage** bootloader: during the first stage it configures memory controllers and SDRAM, while during the second-stage loads

the operating system. As already mentioned, U-Boot also supports DTB hardware descriptions. Before booting the operating system, U-Boot can also provide a **command-line interface** from which several commands can be issued. Those directives can be used to manipulate **flash memory partitions**, load files (such as the Linux kernel) in RAM, set environment variables, etc.

2.4 Development tools

In order to proficiently develop software for an Embedded Linux platform, several development tools must be used. The main difference between **x86** development and embedded (typically **Arm**) development resides in the different architectures. This means that the standard native compiler cannot be used for embedded development.

2.4.1 Cross-compiling toolchains

A compiler that compiles for an architecture different from the host machine one is called **cross-compiler**. For example, a cross-compiler is needed for compiling **Arm** binaries while developing on a **x86** machine. A native Arm compiler could be used on the target system, but this is very impractical to do because often the target device is restricted in terms of available storage and memory. A **cross-compiler** attached to the **Binutils**[1], the **kernel headers**, **libraries** and a **debugger** represent the so-called **cross-toolchain**. Since Arm is one of the most used architectures in the embedded development context, the typical scenario involves using an Arm cross-toolchain (figure 2.4).

The ABI

The ABI (Application Binary Interface) is an interface between two software components that defines **calling conventions** (how function arguments are passed, how values are returned), data alignments and data types. An ABI must be defined when building a toolchain, and all the system binaries should be compiled with the same ABI. In particular, in the context of the **C/C++** languages, the ABI gives standards for implementing the language, ensuring interoperability between parts compiled separately.

Binary stripping

In order to make debugging possible, when compiling a binary, the compiler has to include some **debug symbols**. This is done for enabling the correspondence between the executing binary and the source code it derives from. Debug symbols are not needed when deploying binaries on the target for production, so a **binary**

[1] **Binutils**: a set of tools to generate and manipulate binaries for a given CPU architecture.

stripping operation is executed in order to remove the debug informations from files.

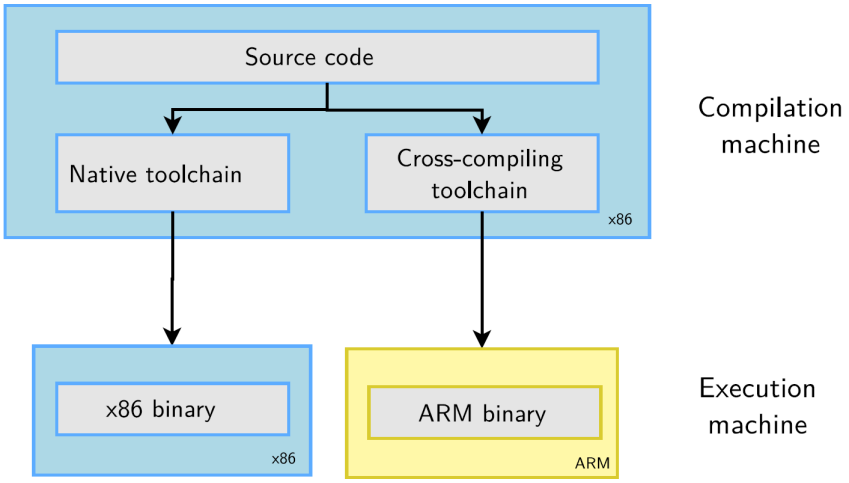


Figure 2.4: The binaries produced from a native toolchain and a cross-toolchain.

2.4.2 Build process management

Building a software unit requires a variable amount of effort, depending on the complexity of the unit. In general, it is possible to directly invoke the **cross-compiler**, or to call `make` for executing automatically a certain number of tasks. For large projects, it is impossible to manually compile and link each component, and it is cumbersome to write by hand a **Makefile** for each unit. For that reason, some utilities like **CMake** and **autotools** are employed.

CMake

CMake is a tool that helps managing the building, testing and packaging processes of software development in large projects. Its main goal is to automatically generate a `Makefile` to be used with `make`. For each project, CMake parses a `CMakeLists.txt` file that can be modified by the user in order to control the build process. CMake supports directory hierarchies, dependencies from multiple libraries and can be fully used with **cross-compilation**.
CMake-based projects are characterised by the following workflow:

1. Write a **CMake toolchain file** that specifies which compiler (cross-compiler) should be used and that, eventually, contains the compilation options to be used.

2. Write the `CMakeLists.txt` file that defines headers, libraries, etc. needed by the project to compile and specifies how the results of the build process should be packaged.

3. In the project folder, call `cmake` passing the toolchain file as a parameter.

4. Since a `Makefile` has been generated, call `make` to complete the build process.

Autotools

The Autotools (also known as GNU Build System) are a set of tools that were created primarily to allow **portability** of software on different **UNIX** systems. As CMake, Autotools work to generate a `Makefile` to be used by `make` and fully support cross-compilation. Autotools consist of **Autoconf**, **Automake** and **Libtool** that are used in order to build a project.

Chapter 3

The Yocto Project

3.1 Introduction

The Yocto Project is an open source set of tools specifically addressed to the creation of custom embedded Linux distributions. It is managed by the Linux Foundation and it is supported by a dedicated community and by several corporate members such as vendors, open source operating systems and hardware manufacturers that share the same purpose: to build Linux-based embedded systems. By sharing the efforts made on embedded products development, each organisation can drastically reduce the time spent on the very same issues: in fact the Yocto Project is completely designed to enhance and encourage modularity, customisation and sharing.

3.1.1 History

The Yocto Project derives directly from another open source project: **OpenEmbedded**. In 2003 some of the developers of the **OpenZaurus** project (a project for the Sharp Zaurus PDAs lineup) founded OpenEmbedded. Their goal was to create a build system for embedded Linux distributions based on a task scheduler inspired by the **Gentoo Portage** package system. This build system was dubbed **BitBake**. OpenEmbedded was used by several organisations, but, due to its uncoordinated development model, it was unsuitable to be used in production environments, giving the difficulty to maintain.
In the meantime, in 2006, the embedded Linux start-up **OpenedHand** created a fork of OpenEmbedded (called **Poky**) that was a cleaner and more supportable version of OE. Poky had a good reputation in the embedded systems field, in fact some of the technical developments of the OpenEmbedded project came via Poky. OpenedHand was acquired by Intel in 2008. OE continuously evolved for supporting more machines and configurations and, after some years, the OpenEmbedded Project became so cumbersome, that the original repository containing the recipes (the main form of metadata to be parsed by BitBake) reached a count of more than

7500 different recipes [8].
For that reason, in 2010, the Yocto Project was founded in the context of the **Linux Foundation**, giving the needed manpower to the OpenEmbedded Project for coherently organising the metadata produced for building software for embedded systems. Poky was also donated by Intel for becoming the reference distribution of the project, thanks to its improvements to the OpenEmbedded build system [11]. Over the next years several organisations joined the Yocto Project Members, thus enhancing the set of metadata ready to be used in the Yocto Project. Some of the most influential companies in the Yocto Project are [18]: Intel, Texas Instruments, NXP, Renesas, Dell, LG Electronics.

3.1.2 The Yocto Project premises definition

The purpose of the Yocto Project can be simply explained in that way [12]: the YP build system takes as input the specifications of the desired Linux embedded distribution and produces as output the main components of an embedded Linux-based OS:

- The Linux kernel for the embedded system
- A `rootfs` containing the desired applications, libraries, utilities, etc.
- A developer **SDK** specifically tailored for the target embedded platform

All of those tasks are performed by the tools contained into the Yocto Project by employing an **agnostic** environment: almost every task that is performed by the YP to achieve the aforementioned results is executed within a freshly compiled build environment, following the Yocto Project internal rules. In this way each utility and executable has always the same version and it is always configured in the same fashion. These premises guarantee that, even with different host build machines, the non-determinism of the building process is kept to the minimum.

3.1.3 Differences between the Yocto Project, Buildroot & OpenWrt

The three most commonly used **build systems** for customised embedded Linux distros are the **YP** (Yocto Project), **Buildroot** and **OpenWrt**. Each of these tools generates a `rootfs` **image** to be deployed on the target, a **kernel image** and a **cross-toolchain**. Nevertheless, there are several differences between these tools (outlined in the table 3.1):

- **Ease of getting started**: OpenWrt and Buildroot are completely based on elements that every embedded Linux developer should be familiar with: **Makefiles**, patches and **Kconfig**. On the other hand, the Yocto Project uses a whole new paradigm represented by **recipes** and **layers**.

- **Industry support**: since the Yocto Project has several corporation members, support is often provided directly from the vendors. Differently, Buildroot and OpenWrt are completely community-based. That difference is a double-edged sword: by using the YP you have complete and costant support by the hardware vendors while, by using OpenWrt and Buildroot, you can generally have more freedom and customisability.

- **Configurability and scalability**: both OpenWrt and Buildroot use **Kconfig** for configuring the builds, which means that every build configuration is contained in one unique file, thus making scalability and expandability to other platforms very cumbersome. The Yocto Project, instead, with its configuration metadata organized in the **Layer Model**, guarantees an high degree of reusability and expandability.

- **Package management**: all the build systems cited above have support to package management, but only the Yocto Project and OpenWrt can natively manage `.rpm` packages (**R**ed Hat **P**ackage **M**anager).

Feature	**The Yocto Project**	**Buildroot**	**OpenWrt**
Ease of getting started	Difficult	Medium	Easy
Industry support	Yes	No	No
Configurability	Yes	Yes	Kind of
Scalability	Yes	No	No
Package management	Yes	Kind of	Kind of

Table 3.1: The main differences between **the Yocto Project**, **Buildroot** & **OpenWrt**.

3.1.4 The Yocto Project architecture

The Yocto Project is an "umbrella" project [20], in a sense that incorporates three main development components:

1. The Poky reference embedded distribution

2. The OpenEmbedded build system (BitBake) and metadata (OpenEmbedded-Core)

3. The Yocto Project integrated tools

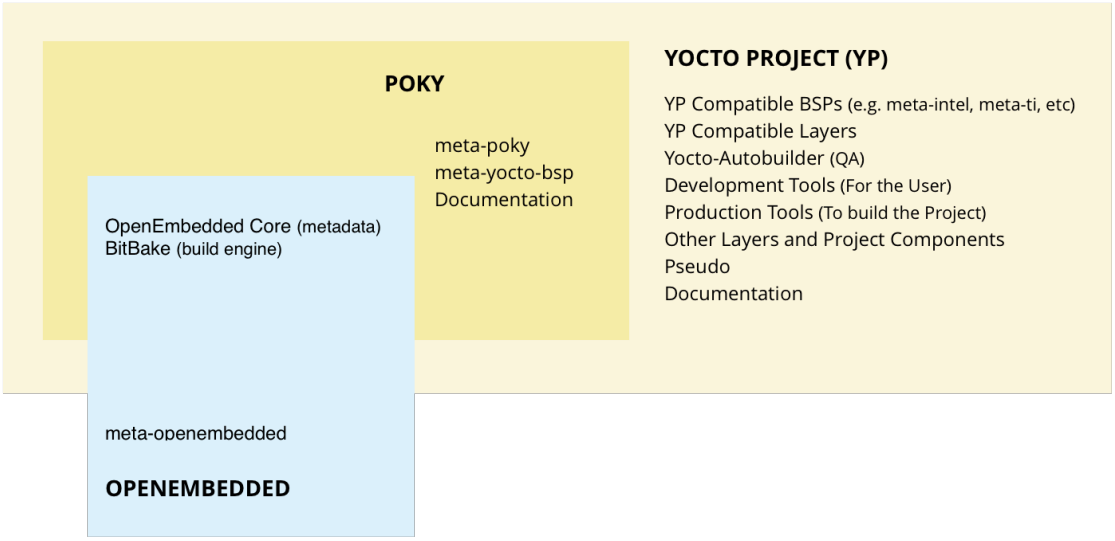


Figure 3.1: The Yocto Project architecture.

3.2 Poky

In the context of the Yocto Project, **Poky** is a term that can mean different things. First of all, Poky is a fork of the original OpenEmbedded project, and it represents the core (containing **OpenEmbedded-Core** and **BitBake**, figure 3.2) of the YP. Poky is also a **reference distribution** of the Yocto Project that can serve as a starting point for creating a custom embedded Linux distribution. Poky has primarily the following purposes:

1. Provide a minimal functional distro used to delineate how to customise a distro.

2. Serve as a test bench for validating and testing the Yocto Project components.

3. Being a self-contained package to be obtained from the users for downloading the Yocto Project.

When you start using the Yocto Project, you download (or **clone**, using the `git` terminology) Poky. Generally, after having defined some of the details about the target machine, you test that everything works correctly by building a minimal image for booting the board (called `core-image-minimal`). What you obtain after this build process is an image containing the **Poky** Linux embedded distribution and an image enclosing a compiled Linux kernel.

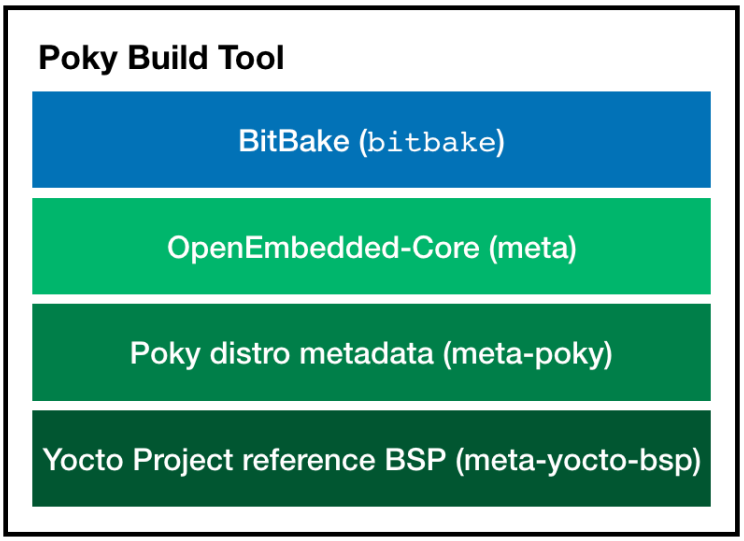


Figure 3.2: The Poky architecture.

3.3 BitBake

BitBake is a `make`-like build tool that is the main core of the Yocto Project build system. BB is a task execution engine that combines both `sh` and **Python** commands in a parallel way, handling complex dependencies between each task. BitBake is completely written in the **Python** language and the parsing unit of BitBake is called a **recipe**: a file that must me be written in a BB-specific syntax.
BitBake is very similar to Make because it controls how software is built, but it can be used for much more complex tasks, such as building an entire embedded Linux distribution. The main differences between `make` and BB are the following:

- BitBake executes tasks by referring to the metadata associated to them. The metadata is stored in recipes and configuration files and contains directives on what tasks to run and the needed dependencies between tasks.
- BB is able to fetch the source code from different places such as local directories, remote servers and source control systems.
- BitBake provides a client/server abstraction that makes it accessible both from command line and from a XML-RPC-based service. Various user interfaces are provided.

3.3.1 BitBake objectives

As already said, BitBake derives from the OpenEmbedded project and extensively utilises the OpenEmbedded-Core metadata. The main BitBake goals that delineated over the years are to:

- Provide extensive support for cross-compilation.
- Handle complex dependency chains between packages.
- Support different kinds of tasks within a package recipe like sources fetching, patching, configuring, etc.
- Being able to interpret tasks written in standard `sh` or in **Python**.
- Be architecture and Linux distro agnostic (for both build and target system).
- Be self contained in a specific part of the build host root filesystem.
- Provide a checksum checking mechanism for improved build speed (shared state).

3.3.2 Shared state cache

One of the best features of BitBake is that it is based on the concept of checksum [1]. A checksum is a particular **signature** calculated from the inputs fed to a task, and it can be used for determining if a certain task needs to be re-executed. Trivially, if the inputs to a task change from one build to another (for example after modifying a recipe), the task needs to be done again, otherwise the already built artifacts can be reused. The **sstate** engine works both with `sh`-based tasks and with **Python**-written tasks.
After verifying that the signature of a task to execute is exactly the same of an already executed one, BitBake starts the **setscene** process. The setscene process allows BB to correctly arrange the pre-built artifacts in the current build context.

3.3.3 The BitBake recipes

As already mentioned, every task that is executed by BitBake is completely descripted in specifc files: **recipes** and **configuration files**. These two kinds of metadata are described in detail in the section 3.6.

3.4 The YP integrated tools

The Yocto Project integrated tools are primarily tools for aiding the development process and for managing the production environment [19]:

3.4.1 SDK & eSDK

The YP provides two Software Development Kit flavours [16]: the **Standard SDK** and the **Extensible SDK (eSDK)**. Like every SDK they mainly include those features:

- **Cross-development toolchain**: it contains a compiler, a debugger and other tools specifically crafted for the target machine and its architecture.
- **Libraries, headers and symbols**: these elements are exactly targeted to the target machine and a chosen image.
- **Environment setup script**: an `sh` script that configures the cross-development environment and correctly sets all the needed variables.

The SDKs produced by the Yocto Project are completely **self-contained**, which means that they include an architecture-specific cross-toolchain and matching **sysroots** for the native and target machines. All of these elements are built by the OpenEmbedded build system, and they are based on the set of metadata that characterises the custom embedded Linux distribution. A great advantage of the Yocto Project-generated SDKs is that, by using these tools, most of the compatibility issues typically encountered in an embedded software development environment are avoided. A common example could be the `gcc` compiler. If a C++ developer using an older `gcc` version doesn't follow the latest C++ language specification, probably nothing would happen, because the compiler wouldn't return any error or warning. If the same application is built by the build engineer (that should deploy the whole embedded software on the target board) using a newer version of `gcc`, the software unit probably would not build correctly, since the compiler is aware of the new language specifications. All of those problems are completely avoided with the Yocto Project SDKs.
Furthermore, both the SDK and the eSDK optionally provide the following components:

- **QEMU** (**Q**uick **EMU**lator): it is a tool that aids the development process by simulating the target hardware on the development machine. QEMU is a hosted virtual machine monitor, which means that it virtualizes the target machine CPU by means of dynamic binary translation. Furthermore, QEMU has support for KVM (Kernel-based Virtual Machine), which allows to use the Linux kernel as a VM hypervisor, thus allowing to run virtual machines at almost native speed.
- **Performance-related tools**: tools like Valgrind, OProfile, RPM, SystemTrap, GCov, GProf, LTTng, etc. that can be used for optimising the development process.

- **Eclipse IDE Yocto Plug-in**: a component for the popular Eclipse open-source IDE that integrates seamlessly the Yocto development workflow into the Eclipse environment.

How the SDK fits into the development process

As already said, the SDKs are directly produced by a complete Yocto Project installation. This means that in a development environment there will be three types of machines: a **Yocto Project Machine**, an **SDK Machine** and the **Target Machine** (figure 3.3). The developers can independently develop applications, kernel, modules, etc. on their SDK machines, and, when their objects are ready for integration, they can be propagated to the Yocto Project maintainer that, on the machine with the full YP installed, can deploy the final image on the embedded system.
In the section 3.8 of this chapter a complete workflow of the development processes with the Yocto Project eSDK is reported, while a manual for producing the SDK on the YP machine can be found in the appendix B.1.

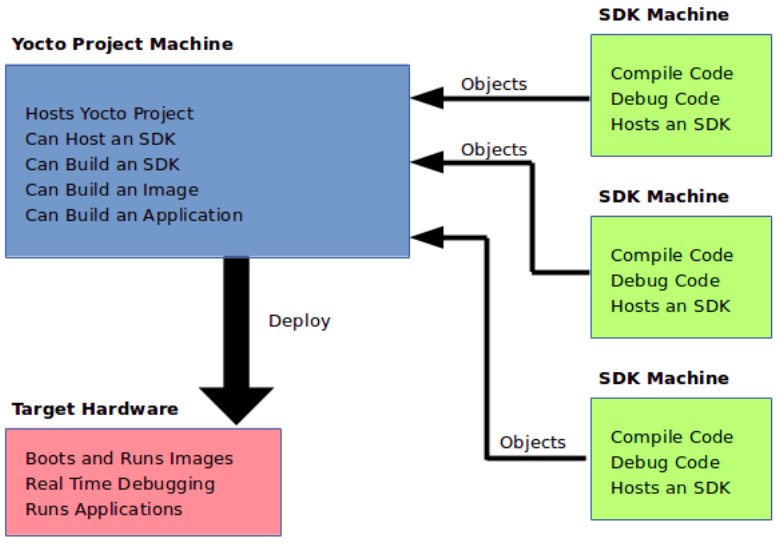


Figure 3.3: The Yocto Project development environment.

Differences between the eSDK and the Standard SDK

The main difference between the two flavours of SDKs is that the eSDK contains tools for adding applications and libraries to an image, directly testing the applications on the target hardware, modifying the source of a component and integrating

the application into the final build system. All of these tasks are performed via a command-line tool called `devtool`. Other differences between the Standard SDK and the eSDK are reported in the following table:

Feature	**Standard SDK**	**eSDK**
Toolchain	Yes	Yes
Debugger	Yes	Yes
Size	100+ MB	1+ GB
`devtool`	No	Yes
Build images	No	Yes
Updateable	No	Yes
Managed sysroot	No	Yes
Installed packages	No	Yes
Construction	Packages	Shared state

Table 3.2: Differences between the Yocto Project Standard SDK and the Extended SDK.

Sysroots

In general, a sysroot is the root directory in which headers and libraries are looked up by development tools. Since, as already mentioned, the Yocto Project (and its derived SDKs) is platform-agnostic, the sysroot is handled differently than traditional SDKs. The YP Standard SDK and eSDK contain in fact two different sysroots: one that is built basing on the root filesystem image of the target board, and one that represents a standardised development environment. This choice guarantees that each developer uses the same version of libraries, compilers, cross-compilers and tools, thus avoiding compatibility issues.
Furthermore, in the case of the eSDK, the sysroots are managed by `devtool`, hence minimizing the possibilities of corrupting the SDK sysroot when trying to add additional libraries.

Shared state tasks

One of the great advantages of the eSDK is that is built upon the concept of BitBake **shared state tasks**. The shared state cache directory containing pre-build objects can be shared among different developers using the Yocto Project eSDK (for example by means of an FTP server). That can aggressively reduce the building times; for example [3], without shared state, a QEMUx86 `core-image-minimal` image takes **35 minutes** to be built, while, with shared state cache enabled, it would take only **1 minute**. Another advantage of shared state construction is that it can be used

directy for adding additional items to the SDK after installation without compiling from source, similarly to a package manager.

`devtool`

The most remarkable added value of the Yocto Project eSDK is `devtool`. `devtool` is a command-line tool that helps developers in building, packaging and testing the software created in the eSDK environment. `devtool` provides support for adding, modifying and upgrading recipes via several subcommands.
The `devtool` has a command line arranged similarly to the `git` CLI (Command Line Interface), with a certain number of sub-commands for each task. The three main subcommands that developers would use in the eSDK environment are:

- `devtool add`: it is used for creating a **BitBake recipe** for building new software.
- `devtool modify`: it uses a previously created recipe for setting up the environment for modifying the source of an existing component.
- `devtool upgrade`: it is used for updating an existing version for building newer versions of source files.

In the section 3.8, a typical workflow using **devtool** is reported.

3.4.2 Toaster

Toaster is a web interface to the Yocto Project build system. It is a friendly instrument for configuring and starting builds and for visualising build informations. Since Toaster can be deployed on a server, it is particularly suitable for configuring a remote build server. The main features of Toaster are the following:

- Project organization: each build performed via Toaster is organised in a project. Each project is characterized by the version of the YP build system.
- Layer browser based on specified layer sources and on the OpenEmbedded Metadata Index. This interface lists images, recipes and machines that are included in each layer.
- Easy steps for adding the needed layers to the project, setting configuration variables and building the selected targets.
- Interface for browsing the directory structure of the built image.
- Interoperability between the BitBake command line and the Toaster web interface.

Furthermore, Toaster stores and provides detailed informations about the build process executed both by the web interface and by BitBake command line (provided that Toaster was up when the tasks started). These are some of the collected statistics:

- What is built and included in the final image
- Error and warnings generated during the build process
- BitBake tasks executed, shared state usage and environment variables set
- Dependency informations between each unit of the build process
- Performance details of the build machine such as CPU usage, etc.

3.4.3 CROPS

CROPS (acronym for **CRO**ss**P**latform**S**) is an open source development framework that uses the **Docker** containers virtualization technology for providing a cross-platform environment (also to) the Yocto Project. CROPS is in fact used for running the YP tools on Windows and macOS hosts. In general, CROPS consists of three components [4]:

- **CREED**: is executed on the development host and exposes an API to Integrated Development Environments (IDEs) or CLI callers.
- **TURFF**: runs in a container and services requests from CODI.
- **CODI** (**CO**ntainer **DI**spatcher): is executed in a container and stores all the executing TURFF instances. CODI transfers the CEED requests to the corresponding TURFF instance.

In the context of the Yocto Project, CROPS offers the support to three containers that are specific to three core components of the YP:

- **poky-container**: an image that, once deployed in a container, drops to a shell specifically configured for installing the full Poky build system (containing all the Yocto Project tools, including BitBake and all the metadata).
- **extsdk-container**: an image that can be configured for automatically installing the Yocto Project Extended Software Development Kit. Once deployed, it drops to a shell in which all the eSDK tools (including `devtool`) can be used.
- **toaster-container**: an image for running the Toaster server in a container.

3.5 OpenEmbedded-Core

OpenEmbedded-Core is a set of metadata (recipes, etc.) that are taken from the original OpenEmbedded Project. After the foundation of the Yocto Project 1.0, the original recipes from OpenEmbedded (called subsequently **OpenEmbedded-Classic**) were divided in several layers for improving supportability. OE-Core is the main Yocto Project layer, in fact it is mantained both by the OpenEmbedded community and by the Yocto Project. This layer contains the core recipes for building an embedded Linux distribution.

3.5.1 Differences between OE-Core and OE-Classic

The main difference between **OpenEmbedded-Classic** and OpenEmbedded-Core is that [10], while OE-Classic represented a giant set of metadata, OE-Core represents the foundation on the top of which machine, application and distribution layers are placed. With OpenEmbedded-Classic, machine and distro-specific overrides were all placed within the same layer, while, with the OE-Core YP model, they should be put in appropriate machine support layers (**BSP** layers) and distro layers respectively.
Another considerable difference between the two OE flavours is that OE-Classic was based on a **push** model on which every developer committed their contributions on the main OpenEmbedded `git` repository, leading often to inconsistent results. OpenEmbedded-Core is instead based on a **pull** model, in which patches are sent to the OE mailing list for review, and than, if considered useful by the maintainer, are merged.

3.5.2 OpenEmbedded-Core scope

The purpose of OE-Core is to provide to the Yocto Project:

- Only recipes needed by almost any configuration and use case
- Support for the main architectures (both 32-bit and 64-bit): ARM, x86, PowerPC and MIPS
- Distro-less environment (as said before, a distro layer should be put on top of OE-Core)
- Support to QEMU emulated machines
- Only the latest version of each recipe

Since older recipe versions are systematically removed from OE-Core, if an older recipe is needed, it should be provided by one of the layers that are added on top of OpenEmbedded-Core.

3.6 The Yocto Project metadata

The metadata is the core concept on which to the whole Yocto Project is built. The metadata is represented by the set of the several configuration files on which all the YP processes are based, and contains informations about all the software that it's used, the commands that need to be executed in order to build each component, the patches that need to be performed on a specific software unit, etc.
In particular, to express all the features of an embedded Linux distributions, several different kinds of metadata are employed in the Yocto Project [20]:

- **Recipe**: file containing several settings and tasks that need to be performed by the build system in order to compile, install and package a required software unit. **Packagegroups** and **images** are specific kind of recipes:
  - **Packagegroup**: a particular kind of recipe that groups the output packages of several recipes into one package.
  - **Image**: the set of packages that should be included in the output `rootfs` to be deployed on the destination machine. An image could also be considered like the entity that represents the desired output of the full build process.
- **Configuration file**: a form of metadata that contains variable definitions used by the build system, hardware configuration infos, etc. Configuration files can be limited to a specific build instance, or could be included in an established set of metadata like the **distro** and **machine** `.conf` files:
  - **Distro**: a configuration file that contains the specific policies employed in the custom embedded Linux distribution to build.
  - **Machine**: settings and parameters that are tied to a specific embedded system and architecture are put in a machine conf file.
- **Layer**: a way of organizing coherent sets of **recipes** and **configuration** files.

It can be noticed that some of the concepts explained above have slightly different meaning within the Yocto Project with respect to the classical Linux terminology that has been presented in the previous chapter.

3.6.1 Recipes

Recipes are the prevailing form of metadata employed through the Yocto Project. A recipe is a `.bb` file that is parsed by the YP build system (BitBake) in order to obtain an output package (or several ones); an example of that would be the compilation of a source file (as specified by the example below) into a binary to be included in the `rootfs` via a specific package. Recipes specify dependencies, patches

to be applied to the source code and detail each step that needs to be performed in order to get the desired output. The tasks can be specified in **Python** or in `sh` standard commands. The essential tasks that are, in order, executed by the build system when parsing a recipe are the following [7]:

1. `do_fetch`: downloads the data from upstream.
2. `do_unpack`: unpacks the downloaded data.
3. `do_patch`: applies patches to the source code.
4. `do_configure`: configures the source tree.
5. `do_compile`: compiles the prepared source code.
6. `do_stage`: installs the compilation results into the staging area.
7. `do_install`: performs the installation into the packaging area.
8. `do_package`: creates a package containing the desired output.

Generally, the only tasks that the user needs to specify in a recipe are the `do_configure`, `do_compile` and `do_install` ones. The remaining tasks are automatically defined by the YP build system. Following there is an example `helloworld` recipe (taken from [17]):

```
SUMMARY = "Simple helloworld application"
SECTION = "examples"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;\
 md5=0835ade698e0bcf8506ecda2f7b4f302"

SRC_URI = "file://helloworld.c"

S = "${WORKDIR}"

do_compile() {
 ${CC} helloworld.c -o helloworld
}

do_install() {
 install -d ${D}${bindir}
 install -m 0755 helloworld ${D}${bindir}
}
```

The most relevant tasks that will be executed when calling `bitbake helloworld` are the following:

1. `do_fetch`: in this case, since the specified `SRC_URI` variable points to a local file, BitBake will simply copy the file in the recipe `WORKDIR`. This is why the `S` environment variable (which represents the source code location) is set to `WORKDIR`.

2. `do_compile`: when executing this task, BB will invoke the C cross-compiler for compiling the `helloworld.c` source file. The results of the compilation will be in the folder pointed by the `B` environment variable (that, in most of the cases, is the same as the `S` folder).

3. `do_install`: this task specifies where the `helloworld` binary should be installed into the `rootfs`. It must be noticed that this installation will only happen within a temporary `rootfs` folder within the recipe `WORKDIR` (pointed by the variable `D`.

4. `do_package`: in this phase the file installed in the directory `D` will be packaged in a package named `helloworld`. This package will be used later from BitBake when eventually building a `rootfs` image containing the `helloworld` recipe package.

Inheritance mechanisms with recipes in BitBake

BitBake provides different ways for sharing common functionalities between recipes. With the mechanisms reported in figure 3.4 a recipe could import tasks and variables defined elsewhere:

- **Class (`.bbclass` files)**: these special recipes could be considered like abstract classes in the object-oriented programming paradigm, in a sense that they contain (task) definitions that can be inherited by other recipes, but cannot be executed directly by BitBake. When a `.bbclass` is inherited, all the functionalities specified in that file are available to the inheriting recipe. It is also possible to inherit multiple classes. A great advantage of the `inherit` directive used for classes is that the inheritance can be performed conditionally by using BitBake overrides or Python scripts.

- **Include (`.inc` files)**: these kind of files are much like an `include` directive in the C programming language. BitBake parses the recipe and, when encounters the `include` or the `require` directive, inserts the content of that file in that location. The main difference between the two directives is that, if the included file is not found by BB, with `include` a warning is returned, while with `require` a parsing error is thrown, thus blocking the build process.

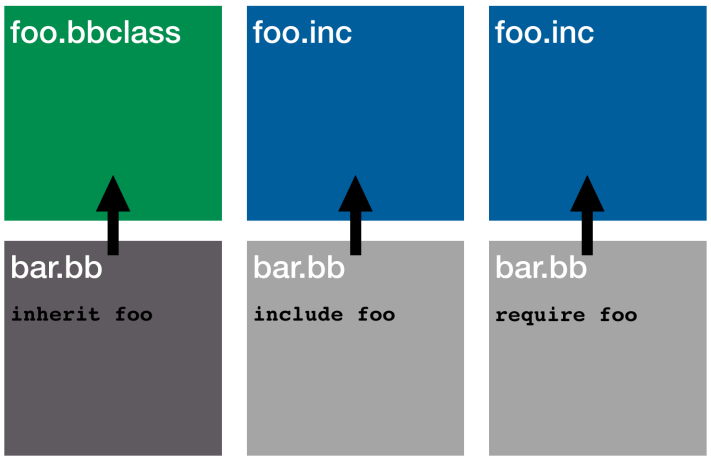


Figure 3.4: The different kinds of recipe inheritance.

Another sharing mechanism offered by BitBake is represented by the `.bbappend` files. Append files extend or override metadata written in an already defined recipe. When BB encounters a `.bbappend` file it expects to find a recipe with exactly the same name of the append file, otherwise a fatal error is returned. This instrument is particularly useful when you need to modify the behaviour of a recipe that is defined in a **layer** that is out of your control (for example one of the Yocto Project core layers). It is in fact extremely dangerous to directly modify already defined recipes, because, in case of future updates of the Yocto Project, all the changes made would be lost.

Common recipes for building software units

Some software units that have common building processes can be standardised by using BitBake `.bbclass` recipes. For example, the Yocto Project completely supports several build paradigms like the **Autotools** (Autoconf, Automake and Libtool), **CMake**, **Make** and direct calling of the compiler (cross-compiler). Each of those systems is handled differently by BitBake:

- **Autotools** & **CMake**-based software: these programming tools, even if substantially different from each other, have in common the purpose of easing the work of the programmer by automating several steps of the building process. The workflows of these tools don't change too heavily when working with an application or another, for that reason in the `oe-core` layer have been defined the recipe classes `cmake.bbclass` and `autotools.bbclass` that contain the common tasks performed when using these build paradigms. These classes, when inherited from a recipe, offload the recipe, almost completely automating the build process. The packaging process of the outputs is also handled autonomously by predefined tasks.

- **Make** and **GCC**-based software: of course it is possible to compile using classic compile methods like using a `Makefile` or directly invoking the cross-compiler. It should be noticed that in the `do_compile` tasks of the recipes `make`, `gcc` and `g++` cannot be called directly, instead the wrappers `oe_runmake`, `${CC}` and `${CXX}` must be used. That is because BitBake automatically generates and passes the right compilation flags to be passed to `make` and **GCC**.

- **The Linux Kernel**: each different flavour of the Linux Kernel has its own BitBake recipe. That is because the source code location is different, but also because each version of the Linux Kernel could require different compilation flags, patches to be applied to the source code, or a different `defconfig` file for configuring the kernel compilation.

- **Kernel modules**: these compilation units require peculiar procedures because, at compilation time, they depend from the Linux source code and the output `.ko` files must be packaged in a specific way. In fact, the kernel module packages should be included automatically when building a **base image** for a compatible **machine**, but shouldn't be included for a **minimal image** that only has the purpose of booting to a working shell. For that reason the recipe class `module.bbclass` handles automatically all of these matters.

GCC has been cited above because it is the default compiler and cross-compiler adopted by the Yocto Project, but support for the **Clang LLVM** front-end for C based languages is provided by the layer `meta-clang`.

The relationship between recipes and packages

Most of the times, a recipe is written in order to perform the compilation, installation on the `rootfs` and packaging of a software unit. Given that, it would be easy to think that building a recipe means to build the package represented by that recipe, but that is not true in the Yocto Project. By default, given the recipe `foo.bb`, the YP build system automatically produces the following packages [9]:

- `foo`: this is the main package originated from the recipe build process. It contains all the library and the binaries obtained through the compilation tasks for the recipe, and any other file needed for running the artifact on the target system. All the binaries in this package are **stripped** from the debug symbols.

- `foo-dbg`: this package contains the debugging informations previously stripped from libraries and executables. By default the `dbg` packages are not deployed on a standard **image**, while are included when enabling debug features.

- `foo-dev`: all files required for development purposes are included in that package, such as shared library symlinks, headers, etc. These packages are very unlikely to be deployed on the target `rootfs`, in fact the `dev` packages are only used for handling dependencies when executing the compiling tasks of depending recipes.
- `foo-staticdev`: same concepts of the `dev` packages, but these packages contain only static libraries.
- `foo-doc`: this package contains documentation files, including `man` pages. These documents are almost never useful on an embedded system, because the end user wouldn't be able to read it anyway.
- `foo-locale`: translation support files are stored in these packages. Generally, only the translations of user applications are included in the `rootfs`.

Package group

Packages can be selectively added to an **image** `rootfs` by using several techniques that will be explained in the following section, but it is cumbersome to keep track of all the packages that are needed in an image. For that reason packages can be grouped in so-called **package groups**. The class recipe `packagegroup.bbclass` handles the package groups and performs the correct generation of all the `dev`, `dbg`, etc. packages. An example of package group is the following:

```
DESCRIPTION = "My Custom Package Groups"

inherit packagegroup

PACKAGES = "\
 packagegroup-custom-apps \
 packagegroup-custom-tools \
 "

RDEPENDS_packagegroup-custom-apps = "\
 dropbear \
 portmap \
 psplash"

RDEPENDS_packagegroup-custom-tools = "\
 oprofile \
 oprofileui-server \
 lttng-tools"
```

```
RRECOMMENDS_packagegroup-custom-tools = "\
 kernel-module-oprofile"
```

This example packagegroup defines two packages called `packagegroup-custom-apps` and `packagegroup-custom-tools` that contain the specified packages, but also the `packagegroup-custom-*-dev` and `packagegroup-custom-*-dbg`, etc. are generated when applicable.

Image

The top-level recipes are the **image** recipes. An **image** is a BitBake recipe that contains all the packages that should be included in the target `rootfs`. These are some of the several images that are defined in the Yocto Project core metadata that can be temporarily customised or inherited for defining new images:

- `core-image-minimal`: a very small image that contains only the required components for booting the target system. It doesn't contain **kernel modules**.
- `core-image-base`: a full console-only image that supports the target system by also including the board **kernel modules**.
- `core-image-sato`: an image that supports the **Sato** graphical environment, particularly suitable to mobile devices. It contains all the device support and some multimedia and productivity tools.

Packages can be added to an already existing image via variable definitions in the `local.conf` file. For example an `example-pkg` could be added to the `core-image-base` image by adding `IMAGE_INSTALL_append_core-image-base = " example-pkg"` variable to the configuration.

3.6.2 Conf files

Configuration files are files that store hardware configuration informations, global declarations and definitions of variables and user-defined ones. In general they give to BitBake clues on what to build and install into the `rootfs` in order to support a particular platform. There are several `.conf` files distributed among the layers of the Yocto Project. Configuration files are the first elements parsed by BitBake during each build process.

`bblayers.conf` & `local.conf` conf files

These configuration files are built by the environment initialisation script that must be executed before invoking BitBake. They are the core files needed by BB for starting the build processes.

- `bblayers.conf` contains all the references to the **layers** that will be scanned by BitBake during the recipes parsing process. A new layer can be added by directly adding the path to its root folder in this file, otherwise it can be done by using the `bitbake-layers add-layer` command.

- `local.conf` is a local configuration file that characterises the current build environment. In this file two crucial variables are defined: `MACHINE` that specifies the target machine, and `DISTRO` that specifies which distro is being used for the target image. Further global variables can be defined in this file, for example `IMAGE_INSTALL_append` for installing other packages into the output image.

Distro

In the Yocto Project context a distro `.conf` file specifies the build configuration policies, high level features of the output Linux embedded distribution, details about the SDK, preferred versions of certain recipes and packages, etc. The most important variables that are defined within the distribution configuration file are the following:

- `DISTRO_FEATURES`: specifies the software support that is needed in the distribution for various features. Usually, a value added to the `DISTRO_FEATURES` variable means that, in the `do_configure` task of some recipes, support for that specific feature will be enabled. An example is the `x11` feature: if added in the distro configuration, the software recipes that support **X11** will compile providing X11 support.

- `PREFERRED_PROVIDER_*`: since a specific functionality could be provided by several recipes, a recipe could add an alias to the `PROVIDES` variable. A typical example of that mechanism is encountered in the Linux **Kernel recipes** that must provide the `virtual/kernel` functionality. The distro configuration file is the right place where the `PREFERRED_PROVIDER_foo` for the example feature `foo` should be specified.

- `PREFERRED_VERSION_*`: this statement must be used when multiple versions of a certain recipe are available, and the build system must be forced to select a specific version.

Machine

This configuration file contains statements and configuration variables that are specific for supporting a target board. These parameters can control each aspect of the whole Linux distro generated for a specific machine such as device options, the `rootfs` image format, etc. Following there are the most common directives specified in a machine `.conf` file.

- `TARGET_ARCH`: specifies the target machine architecture. Some of the architectures supported by the Yocto Project are **ARM**, **x86**, **PowerPC** and several others.
- `PREFERRED_PROVIDER_virtual/kernel`: defines the Linux Kernel that should be built for that machine. As already said, there could be several Linux Kernel versions defined in multiple recipes.
- `MACHINE_FEATURES`: similarly to the variable `DISTRO_FEATURES`, this statement specifies the hardware features that the machine is capable of supporting. Some examples are the `pci`, `bluetooth`, `wifi`, etc. features that could have a direct correspondence to a specific package, or they could simply configure differently certain recipes.
- `KERNEL_DEVICETREE`: specifies the **DTS** files that should be compiled into **DTB** files and included in the Linux Kernel image.
- `KERNEL_IMAGETYPE`: with this variable the Kernel image format is specified. Some of the supported values are `uImage`, `zImage` and `fitImage`. The `fitImage` will include the DTB files specified in the previous variable and the Kernel `zImage`.
- `IMAGE_FSTYPES`: specifies the image format of the `rootfs`. Some of the supported values are `ubi`, `ext3`, `tar.bz2`, etc.

3.6.3 Layers

As already said, one of the great improvements of the Yocto Project over the classic OpenEmbedded project is the classification of metadata into specific **layers**. This paradigm is called the **Layer Model** and it guarantees eased collaboration and customisation. Each layer is logically separated from each other, but the metadata that have been previously defined in a layer can simply be reused and customized in other layers. All the Yocto Project compatible layers are listed and documented in the **OpenEmbedded Layer Index** that can be used as a reference catalogue when looking for new packages (defined in recipes) to include in a project.
The naming convention for the Yocto Project layers is `meta-name`, where `meta` stands for **metadata**. There are mainly three categories of layers that can be defined (figure 3.5), but, if it possible to logically isolate a set of metadata, it is encouraged to create other ones.

The `layer.conf` layer configuration file

This particular configuration file is reported in this section because, even if it is strictly needed by BitBake, doesn't provide any customisation option to the

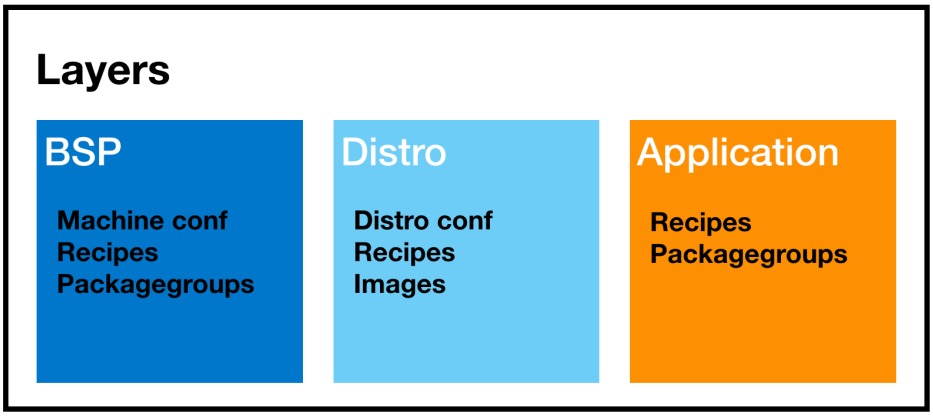


Figure 3.5: The three main types of layers, with all of the metadata that can contain.

user (like the `MACHINE` and `DISTRO` configuration metadata or the `local.conf` and `bblayers.conf` files reported above).
Each Yocto Project layer has a directory called `conf` that contains a file named `layer.conf`. This file incorporates the informations that are needed from BitBake for recognising the layer and the whole set of metadata contained in it. In particular, this file contains the variable `BBPATH` that specifies the root directory of the layer in which BitBake should be able to find all the files inherited by recipes. Furthermore, the `BBFILES` variable specifies the paths in which BitBake expects to find recipes. Dependencies between layers must be expressed instead by the `LAYERDEPENDS_*` variable.

BSP layer

This layer contains all the metadata that characterise machines. Each configuration file and recipe included in this layer should never modify the behaviour of tasks defined in other layers, unless you are building for the specified `MACHINE`. The BitBake mechanisms that are employed to automatically perform this distinction are the following:

- **Folder filtering**: when a recipe `example_1.0.0.bb` includes a local file with the variable `SRC_URI = "file://foo.patch"`, BitBake searches this file in some predefined paths (specified in the `layer.conf` file). If the file `foo.patch` needed by the BB recipe has some machine-specific features when building for `mach1` or for `mach2`, the following solution can be adopted:

```
ooo
└── recipes-example
    ├── example_1.0.0.bb
    └── example
        └── mach1
```

```
│   └── foo.patch
└── mach2
    └── foo.patch
```

This technique guarantees that BitBake will include only the right `foo.patch` file when compiling for a certain `MACHINE`.

- **Variable and task overriding**: by appending `_machine` to a task or a variable, the definition (or override) is executed only for that specific machine. An example would be the definition `VARIABLE_mach1 = foo` or the task `do_task_mach1()`.

The main elements that are found in a BSP layer are the following:

- `MACHINE` conf file
- **Kernel modules** recipes and kernel-related utilities recipes
- **Packagegroups** gathering core packages by topic

Distro layer

The distro layer mainly contains files that define the embedded Linux distribution:

- `DISTRO` conf file
- **Image** recipes that specify the different images of the distribution that can be generated
- `.bbappends` that customise the behaviour of predefined recipes for the custom distribution

Software layer

This layer represents the container of the developed application units and it should contain:

- **Library** recipes for compiling, packaging and installing the libs that are required by the applications
- **Application** recipes for building the apps and specifying their build and runtime dependencies
- **Packagegroups** for organising the application packages in coherent groups

3.7 The general Yocto Project workflow

At this point it is obvious that the Yocto Project is an extremely valuable tool for creating custom embedded Linux distributions. The main issue with the YP is that the learning curve for mastering its components is quite steep; but an organised approach to the initial development phases could dramatically ease these phases. This is not meant to be a guide, but just an overview of the typical workflow for getting started with the Yocto Project.

3.7.1 Getting started

The first task to do is to prepare an host machine for running the Yocto Project. That can be done on a native Linux operating system, on a virtual machine with Linux installed in it, in a cloud-based virtual machine or by using CROPS. After doing that, you should check that you are able to build a **Poky**-based `core-image-minimal` for the `qemux86` (an x86 machine to be emulated by **QEMU**). If everything works fine, the development of the custom distribution can start.

3.7.2 Developing a BSP layer

Before creating a custom distro for a specific target, some hardware-specific metadata must be provided to the Yocto Project build system. Of course, if the target board is a commercial one, it is extremely plausible that a complete BSP layer already exists and could be found via the **OpenEmbedded Layer Index**. If a maintained BSP layer is available, it is strongly recommended to use it; otherwise a custom BSP layer must be developed.
A BSP layer can provide support to **several** different **machines** that have the same architecture in common but differ by other details. Considering a newly create BSP layer targeted to a single board, first of all a `MACHINE` configuration file should be defined. As explained in the 3.6.2 in that file the basic details of the hardware should be specified. After doing that, the tasks that follow should be performed.

Choosing the correct Linux Kernel

Of course, if the target hardware is completely supported by a maintained BSP layer, the kernel choice is quite straightforward, and the vendor documentation should be examined for eventually choosing a different kernel flavour.
If there is the need of creating a custom BSP layer, there are various approaches for selecting the right Linux Kernel for an embedded distro. Several vendors (like **Intel** and **Freescale**) provide BSP layers containing kernel recipes pointing to Linux kernel sources heavily customised for their architectures that could be reused for the custom embedded system. A first approach would be to define `.bbappend` files to these Kernel recipes and apply patches to the kernel source via the BitBake `do_patch`

task, or simply overriding some of the BB predefined tasks. In that way it is possible to pass the kernel configuration delta file (`defconfig`) to the build system, in a way that the built kernel will be correctly configured for the target hardware. Another common objective that can be achieved with the aforementioned method is adding **DTS** files (cited in section 2.1.1) to the `arch/*/boot/dts` folder of the kernel source code so that the Kernel can correctly detect the onboard devices while booting.
If the hardware has too much customisation with respect to the vendor reference designs, a stock Linux Kernel probably wouldn't work correctly on the board. Then, the second approach is to create by scratch a kernel recipe that fetches the customised Linux Kernel sources, likely hosted on a **VCS** (Version Control System).

Defining kernel module recipes

Kernel modules are required for providing full support to all the devices that are available on the hardware. For each device that requires a kernel module a BitBake recipe should be defined. These recipes, as explained in 3.6.1, will produce the `.ko` files starting from the vendor device driver source code.

Creating board support recipes

All the other recipes that help to provide the appropriate hardware support to the distro should be inserted into the BSP layer. A typical example is a recipe for installing the **Udev** (one of the Linux device managers) rules to the `rootfs`, or a recipe for adding **SysVinit** or `systemd` init scripts.

Testing the built BSP layer

After having performed all the previous steps, you should be able to build a minimal working operating system (kernel image + `rootfs`) to test on the embedded system. A straightforward way for testing that everything works correctly is by configuring a build environment (and then editing the `local.conf` file) with `MACHINE` set to the newly defined machine and `DISTRO` set to **Poky**. By building a `core-image-minimal` with BitBake, you should obtain a working Poky distribution for your target board.

3.7.3 Developing an application layer

While the most part of the software layer will be developed after the distribution of the SDK to application developers, it is crucial to create a layer that serves as a skeleton on which building all the app recipes.

Including required library recipes

The **OpenEmbedded Layer Index** contains an huge number of recipes, and it is very likely that any required library is already available. For that reason, sim-

ply putting `DEPENDS += "library"` (if it is a build dependency) or `RDEPENDS += "library-pkg"` (if it is a runtime dependency) inside an application unit recipe should make it compile without any problem. Of course, if a recipe is provided by a particular layer, this must be inserted in the `bblayers.conf` file. On the other hand, if the library is custom-made, or it is an external library not provided by any recipe, a new recipe should be created for building it.

Creating application recipes

As already said, the generation of application recipes should be strictly coupled to the software development process. This means that every recipe of this kind will be generated by using `devtool` when using the Yocto Project **eSDK**.

Generating packagegroups

In order to organise efficiently the inclusion of building artifacts into the `rootfs`, packagegroups should be created by grouping the packages output by the application recipes.

Setting application-specific policies and testing

Some configuration policies could be required for proficiently compiling all the software units. These policies should be included in the `DISTRO` conf file, but, since a custom distribution is yet to be created at this point, all of these options can be reported in the build configuration file `local.conf`. In that way, by using the `IMAGE_INSTALL_append` variable, it is possible to build an image based on **Poky** that contains the developed software in order to test that everything works correctly.

3.7.4 Developing a distro layer

As already said, the distro layer is the place where the shape of your embedded Linux distribution is descripted. It contains informations about package alternative selections, compile-time options, other low-level configurations and describes the different `rootfs` images that can be built.

Setting the configuration policies

All the configuration settings that are required for the application layer to work correctly should be reported here. In general, all the variables that were previously defined in the `local.conf` file should be moved into the `DISTRO` configuration file. A typical example is specifying the wanted implementation of the **C** and **C++** standard libraries.

Choosing different system managers

In the distribution `.conf` file the default device manager and startup daemon can be specified. In particular, with the `VIRTUAL-RUNTIME_dev_manager` variable `udev` or `mdev` can be selected as device managers, while with the `VIRTUAL-RUNTIME_init_manager` definition `sysvinit` or `systemd` can be selected. These two variables use a corresponding mechanism as the `PROVIDES` method explained in 3.6.2.

Creating images

Creating an image substantially means to list every thing that you would like to include on the `rootfs` to deploy on the target development machine. An image recipe could inherit the `core-image-*` images predefined in the **OpenEmbedded-Core** layer (reported in 3.6.1), or could be created by scratch. On both cases, by adding values to the variable `IMAGE_INSTALL`, packages are included into the output image.

3.7.5 Using the Yocto Project as a daily basis

After having successfully designed and tested the several layers that make up an embedded Linux distribution, the whole set of tools included in the Yocto Project can be used for daily development, testing and deploying tasks. In particular, after having built and distributed the **eSDK** based on the defined distribution, all the development work can be performed with the aid of the Yocto Project tools, as reported in the following section.

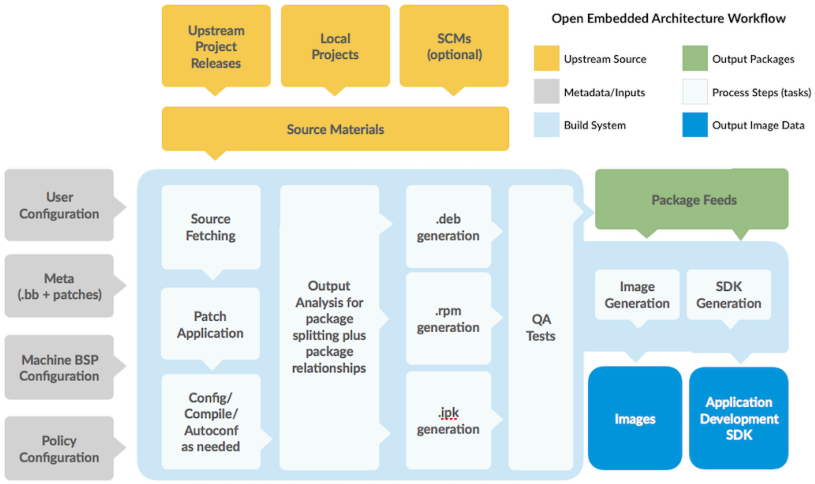


Figure 3.6: The Yocto Project general workflow.

3.8 Developer workflow with the eSDK

After the complete deployment of the YP-based system, with a full conversion of the (eventually) existing building workflow into corresponding BitBake recipes and Yocto Project layers, the development of new software functions (and the maintenance of the already existing ones) can be made by using the **Yocto Extensible Software Development Kit (eSDK)**. The general workflow for developing **new** libraries, applications, etc. ([16] and [15]) can be described as follows:

0. Install the Yocto **eSDK** on the development machine.
1. Create with `devtool add` a new recipe pointing to the new software source code. That could be a local folder on the development machine or a remote repository of one of the several **SCM** tools supported by BitBake like Apache **Subversion**, **Git**, etc.
2. Check that the recipe automatically created by `devtool` has the desired variables and tasks defined in it, otherwise fix them by using `devtool edit-recipe`.
3. Build the newly created recipe with `devtool build`.
4. Test the recipe by deploying the generated package on the target with `devtool deploy-target`, that exploits an SSH server running on the destination machine.
5. If the results are satisfactory, commit the newly created recipe to a permanent layer with `devtool finish`.

After having built (and tested) the new recipe, any modification to it can be conveniently made via the `devtool modify` command. Two comprehensive diagrams depicting the various development workflows supported by the eSDK (taken from [16]) are the 3.7 and the 3.8.

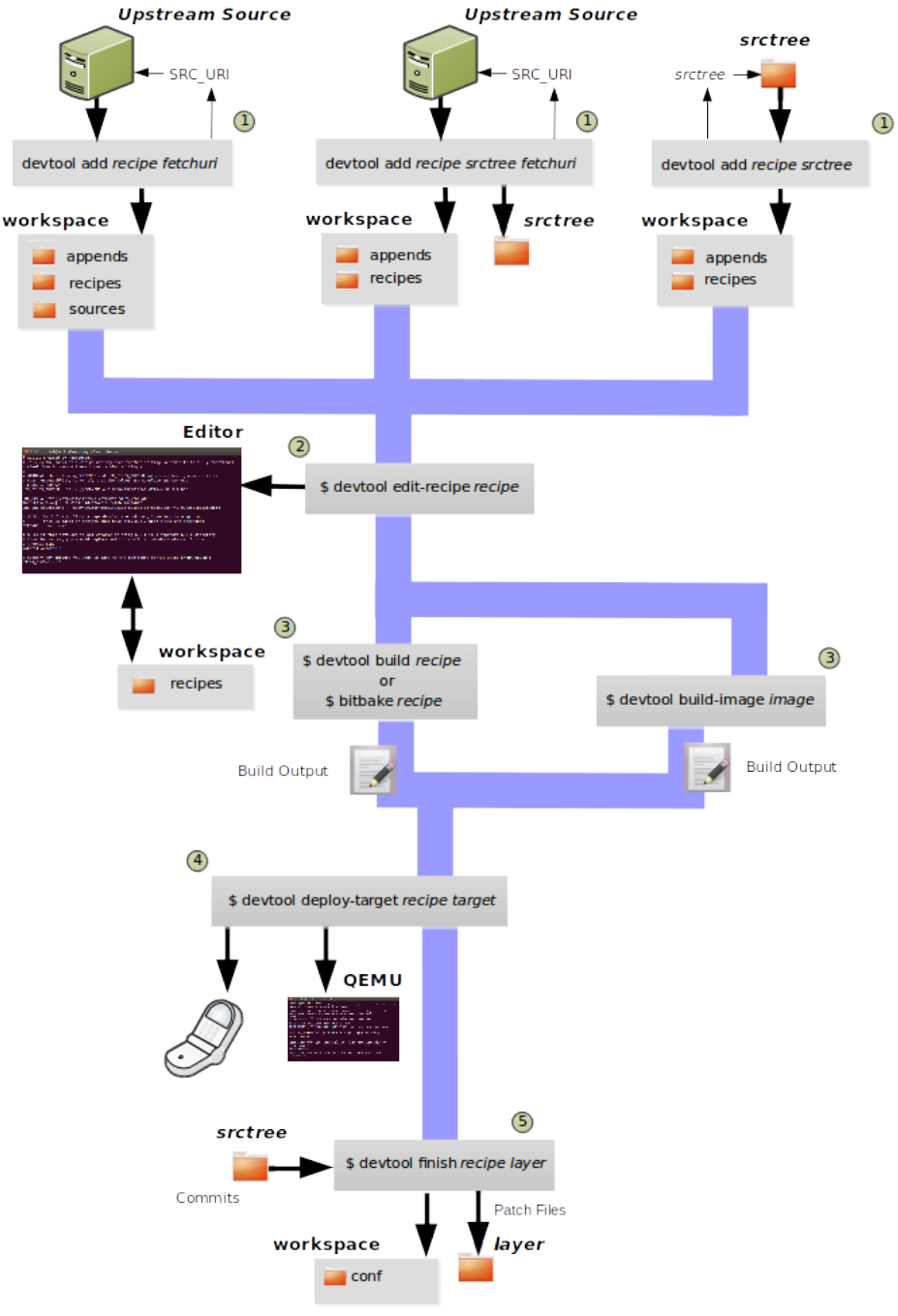


Figure 3.7: Development workflows supported by the Yocto Project eSDK for **adding** a new **recipe**.

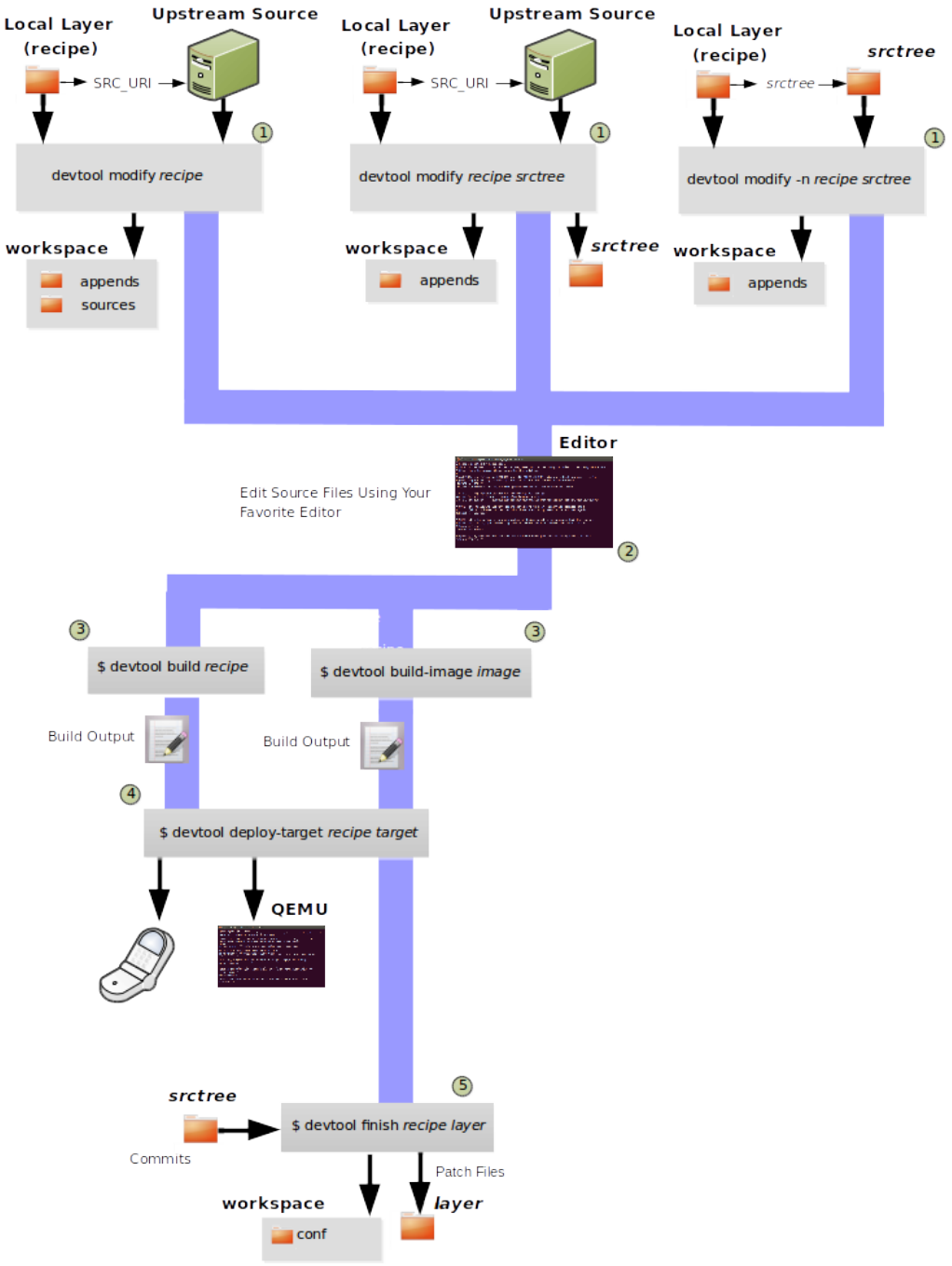


Figure 3.8: Development workflows supported by the Yocto Project eSDK for **modifying** an existing **recipe**.

Chapter 4

The Magneti Marelli Connectivity Framework

4.1 Introduction

This chapter reports some concepts about the **Vehicle-to-Everything** communication and the **Magneti Marelli** Technology Innovation - Innovation & **Connectivity Framework** developed in order to support it. Furthermore, details about the hardware and software facilities employed in the Connectivity Framework project are reported.

4.2 Vehicle-to-Everything

The Vehicle-to-Everything (often referred as **V2X**) communication consists in the transmission of different kinds of messages between **vehicles**, **infrastructures**, **pedestrians**, etc. Different standards are in development for the implementation of this technology, but in this work only the European and North American ones are considered. There are several uses cases that can be made possible thanks to these technologies, and most of them could greatly improve the safety of modern vehicles and pedestrians. In particular, the most common **use cases** that can be implemented by using the V2X standards are the following:

- **Vehicle-to-Vehicle (V2V)**: is the technology that enables automobiles to "**speak**" to each other. In this context the vehicle is often referred as **ITS** (**I**ntelligent **T**ransportation **S**ystem). Some of the use cases that are possible thanks to that type of communication are:
  - **Blind Spot Warning (BSW)**: can inform the driver that another vehicle is located in the blind spot area.

  - **Control Loss Warning (CLW)**: warns other drivers that the vehicle is out of the control of the driver.
  - **Electronic Emergency Brake Light (EEBL)**: a vehicle broadcasts a message saying that it is braking, so that other vehicles receiving the message could avoid a collision.
  - **Forward Collision Warning (FCW)**: warns drivers of a possible upcoming collision with another vehicle ahead in traffic.
  - **Intersection Movement Assist (IMA)**: advises drivers when its hazardous to enter an intersection due to an high collision probability with other vehicles.
  - **Left Turn Assist (LTA)**: warns drivers during an unsafe left turn attempt. That would happen when there is a car approaching on the same path with no intent of stopping.
  - **Stationary Vehicle (SV)**: informs the driver of a stationary vehicle on the same road (a still vehicle with the hazard lights on).

- **Vehicle-to-Infrastructure (V2I)**: is the wireless exchange of messages between vehicles and **road infrastructures** like traffic lights, etc. The most common V2I use cases are the following:
  - **Green Light Optimised Speed Advise (GLOSA)**: the vehicle can provide a suggested speed according to the time it takes to the traffic light to give a green light.
  - **In-Vehicle Road Sign (IVRS)**: the car cockpit can display informations about road signs such as speed limits, warnings, etc.

- **Vehicle-to-Pedestrian (V2P)**: the technology that provides messages exchanging between pedestrians and vehicles, mostly for pedestrian safety purposes (for avoiding the pedestrian to be hit).

- **Vehicle-to-Network (V2N)**: the technology that provides network services and cloud services to the vehicles.

- **Vehicle-to-Device (V2D)**: the connection of smart devices like smartphones, tablets, etc. to the car.

4.3 Technology details

Several wireless technologies are employed in the V2X in order to provide vehicular communication. In particular the following different technologies can be listed:

- **IEEE 802.11p**: this IEEE (Institute of Electrical and Electronics Engineers) standard specifies the wireless frequencies, timings, etc. for enabling Dedicated Short-Range Communications (**DSRC**). 802.11p is based on 802.11 (known as **Wi-Fi**), so it is suitable only for relatively short range applications.
- **C-V2X**: this standard is based on **4G** and **5G** cellular communication technologies in order to provide wider ranges applications.

V2X messages have different specifications and protocols, according to the country in which the technology is deployed. In particular, for the United States and Europe, the defined standards are the following (compared in table 4.1):

- **W**ireless **A**ccess in **V**ehicular **E**nvironments (**WAVE**): is a standard developed by the IEEE (Institute of Electrical and Electronics Engineers), based on the IEEE 802.11p communication standard.
- **E**uropean **T**elecommunications **S**tandards **I**nstitute **I**ntelligent **T**ransport **S**ystems (**ETSI ITS-G5**):

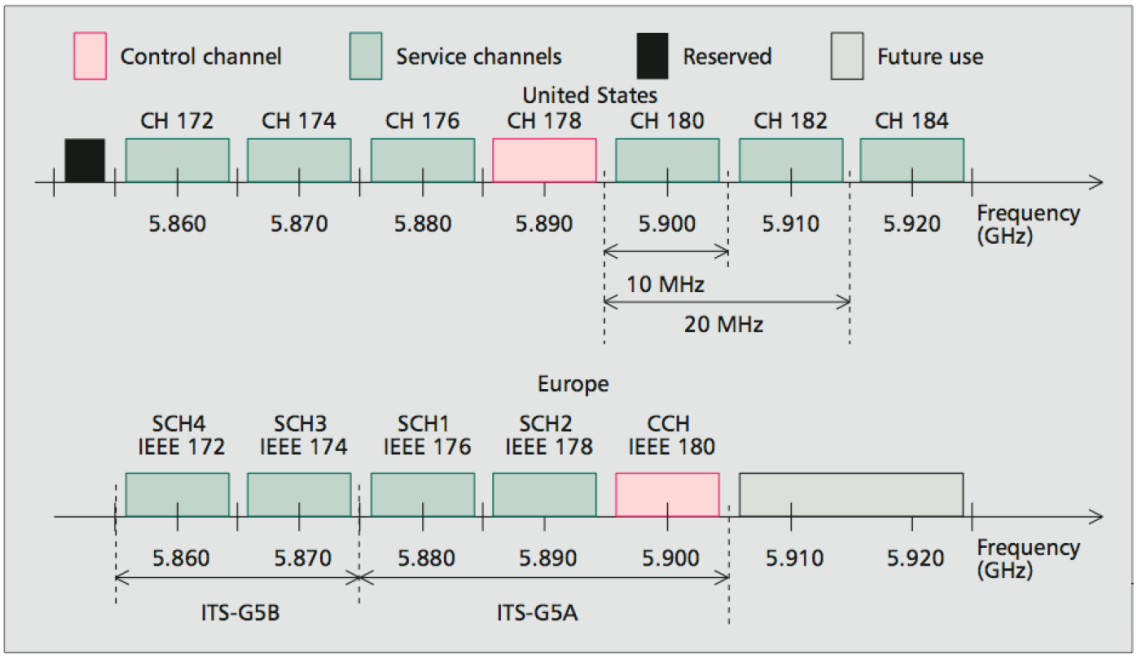


Figure 4.1: The ETSI ITS-G5 and WAVE frequency allocation [5].

4.4 The Magneti Marelli Connectivity Framework

The Technology Innovation - Innovation and Connectivity division of Magneti Marelli is developing a software **Framework** for handling **V2X** communications via **DSRC**,

Specification	**WAVE**	**ETSI ITS-G5**
Bandwidth	75 MHz	50 MHz
Frequency range	5.855 - 5.925 GHz	5.855 - 5.905 GHz
Service channels	6	4
Default MAC support	802.11p	802.11p, Wi-Fi, 4G/5G

Table 4.1: Medium differences between WAVE and ETSI ITS-G5.

C-V2X and compliant with the **WAVE** and **ETSI** standards. The MM Connectivity Framework has a layered structure, characterised by three mian layers: the **middleware**, the **facilities** and the **use cases**. The Framework architecture is reported in figure 4.2.

4.4.1 Connectivity protocol stacks

This layer provides support to the different connectivity technologies used by V2X applications, such as 802.11p or the various C-V2X flavours. Drivers and libraries for transferring informations to/from the devices used by the Framework are placed in this layer.

4.4.2 Facilities

This layer contains a set of software modules and data definitions (such as messages) that provide the required support to the use cases. Each entity can be **common** to the communication standards ETSI and WAVE, or **specific** to one of those. In the following subsections some of those architectural elements are descripted.

Common facilities

The following facilities are **common** between the ETSI and WAVE standards:

- **L**ocal **D**ynamic **M**ap (**LDM**): this entity works as a database that stores relevant data received from vehicles, infrastructures, etc.
- Topology Message (**MAP**): a component that manages a specific kind of message with the same name. This message contains detailed infos about the current road.
- **H**uman **M**achine **I**nterface (**HMI**): it has the responsibility of communicating the informations such as warnings, hazards, etc. to the user (for example by showing them on the vehicle instrument panel display).
- **PO**sitioning **TI**ming (**POTI**): this component provides vehicle location informations such as latitude, longitude and altitude.

- **V**ehicle **D**ata **P**rovider (**VDP**): coupled with the POTI, supplies to the upper layer components other vehicle informations such as its dynamic (coming from the CAN network).

WAVE-specific elements

The following entities are specific to the **WAVE** part of the Framework:

- **B**asic **S**afety **M**essage (**BSM**): a component that handles a kind of message that is periodically sent by the vehicles. The BSM message contains informations about the position, the speed, the dynamics, etc. of the vehicle.
- **T**raveler **I**nformation **M**essage (**TIM**): this component manages messages about traffic, street signs, speed limits, etc.
- **W**AVE **S**ervice **A**dvertisement (**WSA**): this message type represents the announcement of an ITS.

ETSI-specific elements

The following entities are only present in the **ETSI** part of the Framework:

- **D**ecentralized **E**nviromental **N**otification **M**essage (**DENM**): it is a message that is used for warning users of a street hazardous situation.
- **C**ooperative **A**ware **M**essage (**CAM**): this message is similar to the BSM (WAVE). It is sent periodically by the ITS.

4.4.3 The use cases

The MM Connectivity Framework aims to implement all the V2X use cases cited in 4.2 via separate software modules that lie on top of the **common facilities**. This layered architecture (figure 4.2) guarantees that the use case components are not dependent on the underlying technology details (such as DSRC or C-V2X).

4.5 The MM Connectivity hardware

The MM Connectivity Framework has been deployed on several platforms and architectures, but it is mainly targeted to **Arm**-based embedded hardware. The main hardware components that must be available on board to enable the correct functioning of the framework software units are the following:

- **CAN** network for processing vehicle data and for sending it to the car displays.

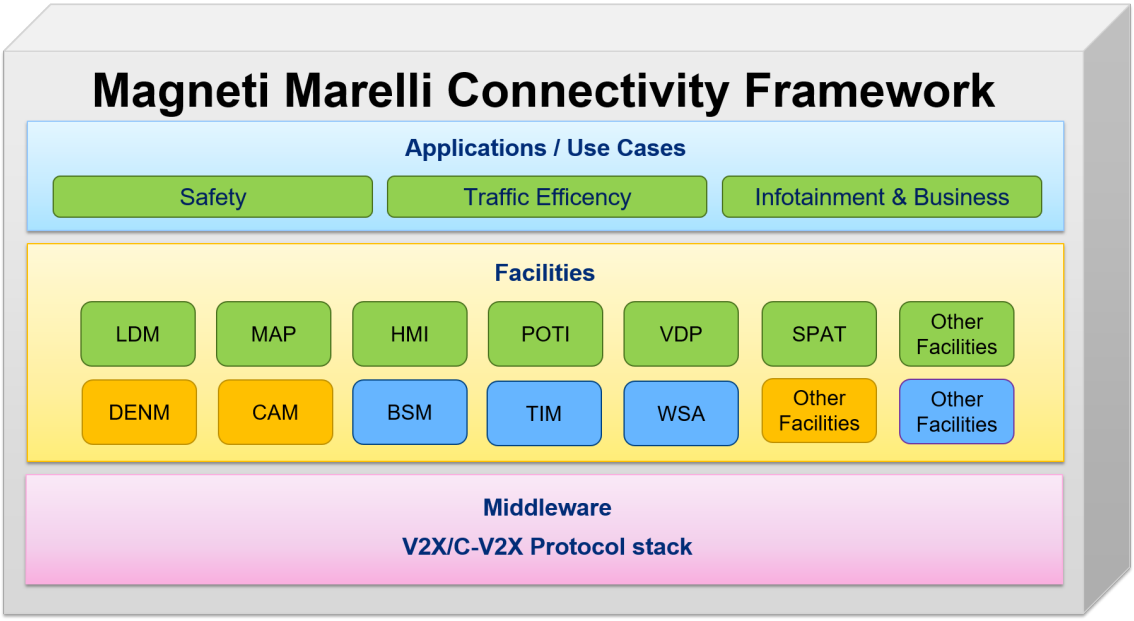


Figure 4.2: The MM V2X Framework architecture. In green the facilities common to ETSI and WAVE standards, in orange the ETSI-specific components, while in blue the WAVE-specific components.

- **GPS/GLONASS** device for obtaining location informations.
- **802.11p** transceiver for enabling the standard V2X communications using the WAVE and ETSI-G5 protocols.
- **4G/5G** modem for working with the C-V2X applications.

Different boards have been used through the development phases of the Framework, but in this work only the **Step 03** proprietary board and an industrial **Car PC** are considered.

4.5.1 The Magneti Marelli Step 03 board

The Step 03 board (figure 4.3) is a Magneti Marelli custom board, developed in the Electronic Systems division. It is based on the NXP-Freescale Smart Application Blueprint for Rapid Engineering (**SABRE**) **i.MX 6QuadPlus** reference design, and it was originally developed for vehicle **infotainment** purposes. The board has been modified for V2X functions and has the following specifications:

- Arm Cortex-A9-based i.MX 6QuadPlus processor
- 1 GB DDR3L RAM

- 512 MB NAND
- 32 GB eMMC
- CAN High Speed & Low Speed
- GPS, Wi-Fi, Bluetooth, 4G modem
- 802.11p transceiver

For a limitation of the i.MX6 6QuadPlus System-on-Chip, the Step 03 supports only two Flexible Controller Area Network (**FlexCAN**) buses. FlexCAN modules provide a full implementation of the CAN protocol specification. In order to communicate with other on-vehicle CAN buses, the PEAK-System PCAN-USB interface is used.

4.5.2 The Car PC

Another device on which the Connectivity Framework is deployed is a Car PC. Car PCs are industrial fanless PCs suitable for in-vehicle usage. The Car PC in question is an **x86** machine with the following specifications:

Figure 4.3: The Step 03 board.

- Intel Core i7-6820EQ processor
- 32 GB DDR3 RAM
- 512 GB SSD (with several expansion slots)
- Wi-Fi, 4G modem

4.6 The MM Connectivity software

The Magneti Marelli Connectivity Framework is completely written in the **C++** programming language. It requires some libraries (and drivers) for interfacing with the onboard peripherals such as the CAN, the GPS, the V2X connectivity devices and the modem. The build process is managed with **CMake**, and all the software source code is uploaded on a **Subversion** repository.

4.6.1 The version control system: Subversion

The version control system used by the MM Connectivity Framework development team is **Apache Subversion**. Each component of the Framework is contained in a module on the Svn repository that is organised in the conventional Svn directory structure:

- `trunk`: this is the main development area. The next release version of the software module should be contained here.
- `branch`: every time the code contained in the `trunk` needs a bug fix or a major change, a new branch should be created. In that way the integrity of the `trunk` can be preserved.
- `tag`: represents a milestone of the software. Most of the times the `tags` contain stable versions of the software module that must be preserved.

4.6.2 The cross-toolchain

The toolchain provided to the Framework developers consists in a VirtualBox **virtual machine** containing the **GCC** cross-compiler, the debugging tools, the required libraries, etc.

4.6.3 The application unit building

Every software unit is built by using **CMake** (and Make). Every application defines a `CMakeLists.txt` file that contains the CMake directives for generating a `Makefile` and correctly compiling the application component. Via the `CMAKE_TOOLCHAIN_FILE` variable, the developer can specify a particular **CMake toolchain file**. Each of those files contains informations on the location of the cross-compiler, the target architecture, the build flags and the directory in which the binaries should be put. The following toolchain files are defined in the context of the MM Connectivity development environment:

- `toolchainArm.*.Debug`: this file specifies the **cross**-compiler location for the **Arm** architecture and enables the **GCC** debug flags.
- `toolchainArm.*.Release`: this file specifies the **cross**-compiler location for the **Arm** architecture **without** enabling the **GCC** debug flags.
- `toolchainX86.*.Debug`: this file specifies the **native** compiler location for the **x86** architecture and enables the **GCC** debug flags.
- `toolchainArm.*.Release`: this file specifies the **native** compiler location for the **Arm** architecture **without** enabling the **GCC** debug flags.

The * in the toolchain file names means that there is a **WAVE** and an **ETSI** version for each of the files listed above. That detail simply means that the output binaries directory is different when compiling for a standard or for another.
Of course, if an application unit requires specific libraries for compiling (or for running), the needed library should be manually compiled (by using CMake or in other ways) and its location should be specified in the `CMakeLists.txt` file. Once the software binaries are obtained, the developers can test their unit by deploying them via `scp`[1] on the target hardware. At this point, if they selected one of the `Debug` toolchain files, debugging can be performed by using `gdb`[2] via an **SSH** connection to the target board.

4.6.4 The building and deploying automation

Since it would be cumbersome to build and deploy the whole Framework component by component with the method described above, a set of `sh` scripts is defined in order to automate the building process of all the Framework components. All the required source code is fetched from the **Subversion** repository of the Framework, from the `tag` directory of each module. There are two different types of scripts based on the technology for which the Framework must be compiled (ETSI or WAVE). These scripts perform the following steps:

1. All the needed libraries and Framework dependencies are compiled by invoking `cmake` and `make` or by using their specific build method.
2. The Framework common facilities are compiled with `cmake`.
3. The Framework ETSI-specific or WAVE-specific components are compiled, based on the script that was started.
4. The Use Cases units are compiled.
5. All the previous results are packaged in a `.tar.gz` archive.
6. The created archive is deployed via `scp` on the target board.

4.6.5 The current development workflow

The development workflow currently employed by Framework developers can be described by the following phases:

- Creation of a new module for containing the software component on the Framework **Subversion** repository.

[1] `scp`: a UNIX tool based on SSH that securely transfers files from a computer to another.
[2] `gdb`: the GNU Debugger.

- Compilation of the libraries needed by the application unit.
- Iteration of the following phases until the development is complete:
  - Coding phase with an IDE chosen by the developer.
  - Compilation of the software unit via `cmake` with debug options enabled.
  - Deploying of the software unit and of the required libraries on the target via `scp`.
  - Debugging via `gdb`.
  - Manual deletion of the test binaries from the target.
  - Commit of the generated code on the `trunk` area of the Subversion module.
- Creation of a `tag` in the Subversion module containing the finished software unit.
- Adding references to the newly created component in the `sh` scripts that compile the **whole** Framework.

4.6.6 Considerations

The methods described above are extremely **easy** and **fast** to use, but, with the constantly increasing complexity of the Framework, several issues could arise:

- It could be difficult to align the version of a specific library between all the developers that use it.
- The `sh` scripts that generate the Framework package must be updated manually with all the right components. That operation is extremely prone to errors.

In the following chapter all the improvements that the Yocto Project could provide to the Framework development team are analysed.

Chapter 5

Development of the `meta-mm` layers

5.1 Introduction

As described in the previous chapter, **The Magneti Marelli Connectivity** projects are characterised by an increasing complexity and constantly evolving specifications. For that reason, the Yocto Project is the perfect match for handling seamlessly and, most of all, automatically the build and integration processes. The following chapter explains in detail all the choices that have been taken in order to integrate the Yocto Project into the existing MM Connectivity working environment, including the required changes to the current workflow that need to be employed.

5.2 Advantages of the YP within the MM Connectivity environment

As already mentioned, the Yocto Project could greatly improve the workflow and the development environment of the projects of the Magneti Marelli TIIC division. These are the main improvements (summarised in the table 5.1):

- **Learning curve**: the Yocto Project offers a complete set of tools that are based on a paradigm that is quite different from the one used from the development team. This means that adapting to the YP eSDK workflow requires a self-education effort. An typical example of that is the cross-compiler: while it is still possible to invoke the cross-compiler within the eSDK generated `rootfs`, using `devtool` for compiling software units is the recommended method.

- **Focus on business logic**: after mastering the Yocto Project and completely arranging the development environment, the focus of the developers can be completely oriented towards the business logic of the software, and not to the build process details.

- **Unified cross-compiler**: as already stated before, using non-aligned cross-compiler versions could lead to severe integration problems. During the development of the Magneti Marelli **application layer**, that exact issue was encountered with one of the core software units of the Framework. Of course, if the aforementioned software unit was developed using the eSDK from the start, that issue wouldn't have occurred.

- **Standardised libraries**: when a new library is needed from a developer, it must be integrated into his SDK so that he can work on his application. There is no guarantee that other developers, if needing the same library, will download the same library version. Of course that issue could be solved with proper communication and documentation between the developers, but the Yocto Project eSDK completely eradicates the issue. In fact, the most commonly used libraries are all available through the **OpenEmbedded Layer Index**, and, for a specific Yocto Project version, will always have the same version.

- **Easy updatability**: updating compilers, libraries, utilities, etc. would be extremely difficult in the TIIC development environment because that would require to update each component separately and then redistribute a new SDK among the developers. The YP, instead, updates all of the maintained recipes for each release, so updating the Yocto Project and redistributing the eSDK would automatically update all the included software.

- **Platform agnostic**: the TIIC SDK is distributed in the form of a VirtualBox Ubuntu virtual machine containing the operating system and the whole SDK. With the Yocto Project the eSDK is self-contained and can be installed easily on several Linux distributions and, by using **CROPS**, could be even deployed on a Docker container running on Windows or macOS.

- **IDE integration**: by using the Eclipse IDE with the Yocto Project plug-in installed, all the development phases (coding, building, debugging, deploying) can be performed via the Eclipse GUI. Of course, if another IDE is chosen, all the development tools are always accessible by the `devtool` CLI. The TIIC SDK, instead, gives you complete freedom on the IDE choice, but it forces you to use the CLI toolchain components.

- **Web interface**: the YP contains **Toaster**, that can be used for managing remote builds with an easy user interface, even without a deep knowledge of how the Yocto Project build system works.

- **Deterministic image generation**: being BitBake completely based on the concept of rules (recipes) to follow, the output `rootfs` images produced with it contain only and exactly the packages that have been specified in the dedicated

recipe. The TIIC SDK doesn't have a standardised way of producing images to deploy on the target hardware, because the packages are inserted into a pre-built core image by using a set of `sh` scripts.

Feature	**Yocto Project**	**Standard MM TIIC workflow**
Learning curve	Steep	Gentle
Focus on business logic	Complete	Partial
Unified cross-compiler	Yes	No
Standardised libraries	Yes	No
Easy updatability	Yes	No
Platform agnostic	Yes	No
IDE integration	Eclipse	No
Web interface	Yes	No
Deterministic image generation	Yes	No

Table 5.1: Differences between the Yocto Project workflow and the existing Magneti Marelli Connectivity projects workflow.

5.3 Development process

In the following section the development process of the Yocto Project support to the Magneti Marelli Connectivity Framework is reported. This work is mainly focused on the **802.11p WAVE** flavour of the Framework, but can be easily extended to other versions of the Framework.

5.3.1 First phase: the BSP

The first goal of this work was to create a Yocto Project-compatible Board Support Package (BSP) for the Magneti Marelli Step 03 board. Since the Step 03 is based on the NXP-Freescale i.MX 6QuadPlus SABRE-AI reference design, the starting point for the Step 03 BSP was the `MACHINE imx6qdlsabreauto`, contained in the layer `meta-freescale`.

The Linux kernel for the Step 03

The phases for selecting a proper Linux kernel for the Step 03 TIIC distribution support were executed in a **bottom-up** fashion, starting from an almost **vanilla Freescale Linux** kernel up to an highly customised kernel. The motivations of this

choice are further explained in the subsection 5.6.1. These are the main steps that were taken:

1. The first step was to try to build a Poky-based `core-image-minimal` with the default Linux kernel for the `imx6qdlsabreauto` to see if the operating system was booted correctly. The default Kernel (dubbed `linux-fslc`) is a fork of the original Linux Kernel, and its default version on the **Rocko**[1]-compatible `meta-freescale` is the 4.1. Unfortunately the Step 03 was **not booting** with that configuration because the kernel was not able to recognise all the onboard devices.

2. The second step was then to include the correct Device Tree Source (DTS) files in the `linux-fslc` kernel. As explained in 3.6.2 and 3.7.2, DTS files can be included in the kernel sources via patches or by specifying BitBake task appends. Even if the kernel was now recognising some of the needed devices, the boot process was stuck at a certain point. It became clear that the `linux-fslc` kernel was too generic for the highly customised Step 03 board.

3. The third step was then to obtain from the `git` repository of the original Step 03 hardware/software designers team the Linux Kernel 4.1.15 source code customised for the Step 03. This kernel was developed for the board (targeted to infotainment purposes) from which the Step 03 derives. This source code was integrated into the TIIC **Subversion** repository, and then a new recipe for building this kernel was created in a new BSP layer. This recipe was written for automatically fetching the kernel source code from the repository and for correctly compiling the kernel. By using also a Step 03-specific `defconfig` file, the board **successfully** completed the boot process.

The table 5.2 summarises the three steps performed before obtaining a working Linux Kernel.

The `imx6qpstep3` machine

After having successfully tested a minimal `core-image-minimal rootfs` based on the `imx6qpsabreauto` machine, some tweaks related to the Step 03 were consolidated in the `imx6qpstep3.conf` **machine configuration** file:

```
#
# Machine configuration for the Magneti Marelli i.MX6QP Step 03
# Type: MACHINE
# SOC: i.MX6QP
```

[1]Rocko: codename of the Yocto Project 2.4 release. The `meta-freescale` layer releases, like every other maintained layer, follow the Yocto Project release cadence.

Linux Kernel	**Kernel configuration**	**DTS files**	**Booting process**
`linux-fslc` 4.1	Default configuration	Default `imx6qdlsabreauto` DTS files	Not booting
`linux-fslc` 4.1	Default configuration	Custom Step 03 DTS files	Hung
Custom Linux Kernel 4.1.15	Custom Step 03 `defconfig` file	Custom Step 03 DTS files	Booting

Table 5.2: The different Linux Kernel flavours tested on the Step 03 board.

```
# Maintainer: Alessandro Flaminio
# <alessandro.flaminio@external.magnetimarelli.com>
# Copyright (C) 2018 Magneti Marelli S.p.A.
#

MACHINEOVERRIDES =. "mx6:mx6q:mx6dl:"

require conf/machine/include/imx-base.inc
require conf/machine/include/tune-cortexa9.inc

PREFERRED_PROVIDER_virtual/kernel = "linux-mm"

# An UBI volume is generated (to be flashed by using ubiformat)
IMAGE_FSTYPES = "ubi"
# The following parameters are specific to the Step 03 MTD partitions
MKUBIFS_ARGS = " -m 2048 -e 126976 -c 3920 "
UBINIZE_ARGS = " -m 2048 -p 128KiB -s 2048 "

# Changes the volume name of the rootfs
UBI_VOLNAME = "ev_fs"

KERNEL_DEVICETREE = " \
imx6qp-g25-prs1-1280x480.dtb \
"

# A kernel fitImage is generated
KERNEL_CLASSES += "kernel-fitimage"
KERNEL_IMAGETYPE = "fitImage"

# In case a not minimal image is built, the packages containing
```

```
# kernel modules, Udev rules, etc. are included
MACHINE_EXTRA_RRECOMMENDS += "packagegroup-mm-step3"

. . .
```

These are the most important details included in the `imx6qpstep3.conf` configuration file:

- The variable `PREFERRED_PROVIDER_virtual/kernel` specifies that the Linux kernel to be used with the Step 03 is the customised Magneti Marelli kernel cited previously.
- The variables `IMAGE_FSTYPES`, `MKUBIFS_ARGS` and `UBINIZE_ARGS` are used for correctly generating the **UBI** volume image of the `rootfs` that must be flashed on the NAND.
- `KERNEL_DEVICETREE` specifies which **DTS** should be compiled in **DTB** by the `dtc` compiler.
- The variable `KERNEL_IMAGETYPE` defines the kernel image to be generated.
- `MACHINE_EXTRA_RRECOMMENDS` is used in order to automatically include certain packages when building a non-minimal image. In this case, the packagegroup `packagegroup-mm-step3`, as explained later, contains some Step 03 BSP components.

Kernel modules for the Step 03

Since, as explained in 4.5.1, the Step 03 board is equipped with an 802.11p transceiver and should support external **PCAN-USB** adapters, some kernel modules are required for correctly interfacing the Linux kernel and applications with these devices. Principally there are two drivers for which have been defined two different recipes:

- **PEAK-System PCAN-USB driver**: the recipe for building this kernel module instructs BitBake to fetch the driver from the PEAK-System website, apply a patch to the module `Makefile`, and package the outputs of the building process. The `Makefile` patch is needed because the building process requires both the Linux Kernel source code and the output artefacts of the Kernel compilation. Since the YP build system stores these files in different locations, some of the environment variables contained in the `Makefile` needed to be changed.
- **Cohda Wireless V2X driver**: since this driver cannot be directly obtained from the web, the defined recipe is coupled with a `tar.gz` archive containing the source code. In this recipe the binary stripping process (reported in

2.4.1) performed by the Yocto Project build system was disabled because the kernel object, once stripped from its debug symbols, somehow was not loaded correctly by the operating system.

Udev rules for the Step 03

In order to make the `gpsd` GPS daemon recognise the GPS serial interface, a Udev rule needs to be defined. This rule specifies the Step 03 serial interface to bind to the GSP daemon. The recipe developed for this purpose packages the `.rule` file in order to be installed on the `rootfs` in the right location (`/etc/udev/rules.d`).

5.3.2 Second phase: the Magneti Marelli Connectivity Framework and applications

After correctly deploying a minimal `rootfs` on the Step 03 and having tested all the needed peripherals, the focus of this work shifted towards the creation of a set of metadata for building and packaging the MM Connectivity Framework.

Miscellaneous tools

First of all, some basic application tools such as an SSH server, network tools, GPS tools, etc. needed to be included into the `rootfs`. For adding those tools into the filesystem these are the general steps that were followed:

1. Finding on the **OpenEmbedded Layer Index** which layer was providing a recipe containing the required package.

2. Eventually adding the needed layer in the `bblayers.conf` file.

3. Building and including the required binaries into the output image via the `IMAGE_INSTALL_append` variable.

In the following section is explained how those tools were packaged in specific packagegroups to be included in newly defined images.

The libraries and applications

New BitBake recipes were defined for each proprietary library and application component of the Framework. Since every software unit of the Framework is built by using `cmake`, the `cmake.bbclass` class recipe was used extensively. Considering that the Yocto Project automatically manages the **toolchain**, there is no need to pass extra variables to `cmake` in order to correctly locate and configure the cross-compiler, etc. That is a big difference with the toolchain configuration explained in 4.6.3, and for that reason some adaptations needed to be made. In particular, **all**

the CMakeLists.txt of the Framework components contain some variables (such as specifications about the C++11-compliant compiler, etc.) that conflicted with the configurations automatically performed by the Yocto Project build system. There were two options: modifying all the CMakeLists.txt directly on the Svn repository or patching all the CMakeLists.txt in each software unit recipe. The second choice was made, because, in that way, the compatibility with the older build system was not broken.
Furthermore, all the dependencies from open source libraries were handled by adding the appropriate values to the DEPENDS variable, thus taking advantage of all the recipes contained in the **OpenEmbedded Layer Index**.
Below there is the recipe (called libutility_3.0.6.bb) for building the libUtility library of the Connectivity Framework:

```
#
# Recipe for building the libUtility
# Maintainer: Alessandro Flaminio
# <alessandro.flaminio@external.magnetimarelli.com>
# Copyright (C) 2018 Magneti Marelli S.p.A.
#

LICENSE = "CLOSED"
LIC_FILES_CHKSUM = ""

# NOTE: the fetcher will work only if you have saved your Subversion
# username and password in the variables SVN_USER and SVN_PSWD.
# You can define these variables in local.conf.
#
# SRCREV is the revision of the source code to checkout.
# The version of the recipe is used for fetching the correct
# source code tag.

SRC_URI = "${SVN_PATH}/10_Framework_src/common/libUtility/tags/;\
           module=libUtility-${PV};\
           protocol=http;path_spec=${PN}-${PV};\
           user=${SVN_USER};pswd=${SVN_PSWD}\
           "
SRCREV = "7223"
SRC_URI[md5sum] = "51e5fb8d2f1500fe45ec8987d97fc865"

SRC_URI += " file://0001-fix-CMakeLists.patch"

inherit cmake
inherit mm-cmake
```

```
DEPENDS += "boost libconfig"

# Specify any options you want to pass to cmake using EXTRA_OECMAKE:
EXTRA_OECMAKE = ""
```

In particular, some interesting details about the reported recipe are:

- The source code of the `libUtility` is fetched by the BitBake **Svn** fetcher from the Framework repository. The variables `$SVN_PATH`, `$SVN_USER` and `$SVN_PSWD` are all custom defined, and should be assigned in the `local.conf` file. These variables represent, respectively, the path of the Svn repository, the Svn username and the Svn password.
- The patch `0001-fix-CMakeLists.patch` is applied before of the build process to the `CMakeListst.txt` file for the reasons explained above.
- The recipe inherits the class `mm-cmake` that contains some CMake variable assignments common to all the Framework recipes.
- The `DEPENDS` statement specifies that the recipe, during the **build** process, depends on the **Boost** and **libconfig** library recipes.

The C++11 ABI issue

The C++11 standard issued in 2011 extensively modified the C++ language with additions and modifications. One of the main changes was made in the implementations of the `std::string` and `std::list` classes; in fact C++11 forbids the Copy-On-Write strings and requires to keep track of string sizes in lists [13]. This implementation changes led the **GNU** `libstdc++` to change its **ABI** (**A**pplication **B**inary **I**nterface) starting from GCC version 5.1.
The Magneti Marelli Connectivity Framework makes use of a proprietary V2X library: the **Marben V2X stack**. This library is distributed pre-compiled using GCC 4.8, thus using the older `libstdc++` ABI. In order to be able to correctly compile the Framework with the Yocto Project Rocko default GCC version (**7.3.0**), a **macro** needed to be passed to the compiler so that the **old** ABI was used. This macro (`_GLIBCXX_USE_CXX11_ABI=0`) was included in the `TARGET_CXXFLAGS` BitBake variable in the `local.conf` configuration file.

5.3.3 Third phase: the TIIC distro

After having built and tested the Linux **kernel**, the **BSP**, the **`rootfs`** minimal contents and the whole Magneti Marelli Connectivity **Framework**, the wrapping up phase of this work was to define the new **Distribution**.

Definition of the tiic.conf DISTRO configuration file

In this file the basic features of the distro were defined such as ipsec and ssh-server-dropbear to provide support to IPsec and the SSH server Dropbear respectively. Furthermore, the GCC macro _GLIBCXX_USE_CXX11_ABI=0 cited above was also included in the tiic.conf file for correctly compiling the Framework.

Definition of packagegroups

In order to efficiently keep track of all the defined packages, the following packagegroups where defined:

- packagegroup-mm-step3: this packagegroup contains the BSP for the Step 03 board. It was in fact inserted in the variable MACHINE_EXTRA_RRECOMMENDS in the imx6qpstep3.conf MACHINE configuration file.

- packagegroup-mm-tools: encloses the basic tools needed by the TIIC. This packagegroup is machine-independent.

- packagegroup-mm-connectivity-common: this packagegroup contains the packages produced by building the recipes related to the Framework common facilities.

- packagegroup-mm-connectivity-wave: consists of all the built WAVE-specific Framework components.

- packagegroup-mm-connectivity-config: encloses all packages containing Framework-related sh scripts and configuration files for supporting different vehicles.

Definition of images

The following distro images have been defined:

- core-image-mm: this is a basic image based on the Poky core-image-base. This image contains a BusyBox-based set of utilities and the contents of the packagegroup-mm-tools package. If the selected machine is imx6qpstep3, also packagegroup-mm-step3 is included.

- mm-image-v2x-wave-all: this image is based on the core-image-mm and includes all the WAVE V2X (DSRC) Connectivity Framework software.

5.3.4 Fourth phase: x86 porting of the TIIC distro

After having developed and tested the **TIIC distro** for the **Step 03** board, a further step was to port the distro on a **x86 Car PC** and configure dual-booting with an existing **Ubuntu** installation. Given the great scalability of the Yocto Project, generating the distribution image for another architecture required only a limited amount of effort.

Intel metadata

Even if the `meta-yocto-bsp` provides out-of-the-box support to the `genericx86` machine, the `meta-intel` layer guarantees a better support for **x86 Intel CPUs**. `meta-intel` provides a few carefully selected tune options and generic hardware support to cover the majority of current Intel CPUs and devices. Furthermore, the kernel recipe `linux-intel` is included, that brings better Intel hardware support to the current LTS (Long Term Support) Linux kernel. Since the Car PC has an **Intel Core i7** CPU, the machine `intel-corei7-64` was selected in the `local.conf` file.

Image generation

Exactly like the build process for the `imx6qpstep3`, the image recipe `mm-image-v2x-wave-all` can be used with **TIIC** to generate an image for `intel-corei7-64`. The main difference between the two processes is that, when building for Arm-based devices, a kernel image and a `rootfs` image are generated, while, for an x86 system, the recipes contain rules for generating a `.wic` image that consists of the complete Linux distro. This difference is explained by one main distinction between general purpose systems and embedded (Arm-based) ones: embedded devices have integrated memories, while PCs have HDDs/SSDs.

Dual-booting the TIIC distro and Ubuntu

The `.wic` generated image is compatible with the **UEFI** (Unified Extensible Firmware Interface), in fact it consists of two partitions, one containing the UEFI data and the kernel image and one containing the `rootfs`. This means that, in order to configure a dual-booting of the pre-existing Ubuntu installation with the TIIC distro modifications to the bootloader and to the partitioning of the hard drive needed to be performed. In particular:

1. A new partition was created on the hard drive in order to accomodate the TIIC `rootfs` partition of the `.wic` image.

2. The UEFI partition of the Car PC was modified by including in it the generated Linux **kernel image** and the **Intel microcode**.

3. A new entry to the **GRUB bootloader** (the default bootloader included with Ubuntu) was created in order to correctly load the newly installed TIIC distro.

5.4 Layers

All the work described above was organised by using the Yocto Project **Layer Model**. In particular, the three typical layer types were defined: BSP, distro and application layers.

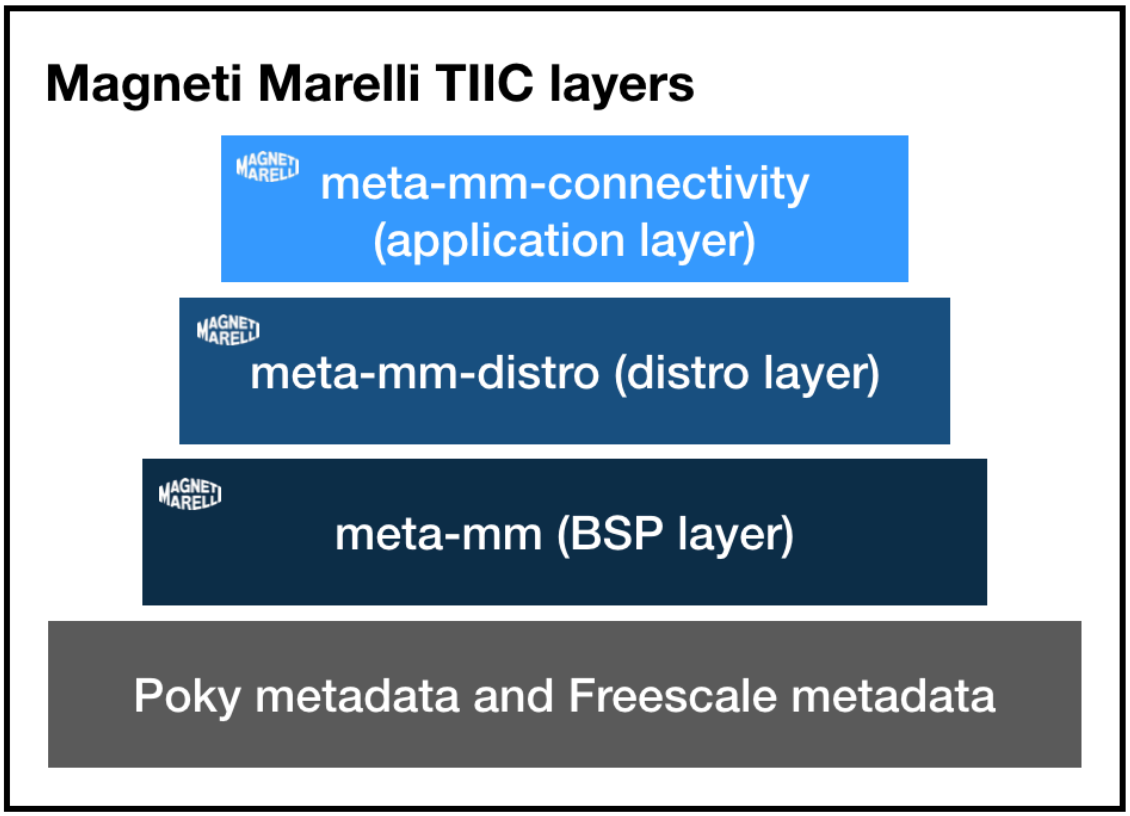


Figure 5.1: The developed Yocto Project layers for Magneti Marelli TIIC (Technology Innovation - Innovation Connectivity).

5.4.1 `meta-mm`

This is the **BSP** layer and it is organised as reported in figure 5.2. The complete directory tree of the metadata contained in `meta-mm` can be found in appendix C.1.

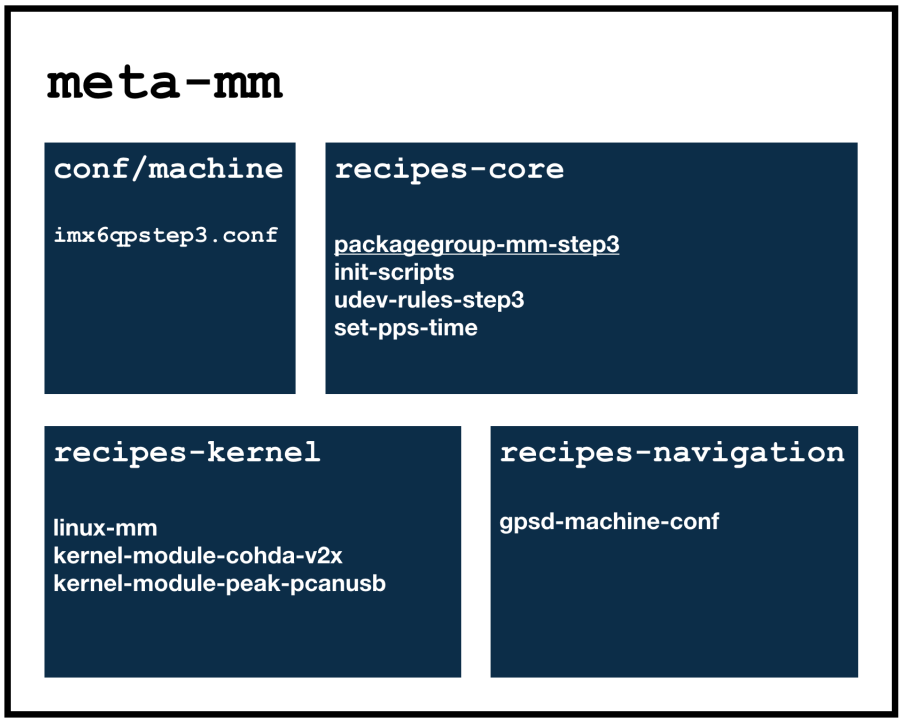


Figure 5.2: The recipes arrangement in the BSP `meta-mm` layer.

5.4.2 `meta-mm-distro`

This is the **distro** layer and it is organised as reported in figure 5.3. The complete directory tree of the metadata contained in `meta-mm-distro` can be found in appendix C.3.

5.4.3 `meta-mm-connectivity`

This is the first **application** layer defined for the Magneti Marelli Connectivity Framework. It contains all the V2X-related recipes (figure 5.4). The complete directory tree of the metadata contained in `meta-mm-connectivity` can be found in appendix C.2.

5.5 The eSDK and the new YP-based workflow in the Magneti Marelli Connectivity environment

As already said, the deployment of the Yocto Project in a development environment significantly changes (in better) the workflow. First of all, for starting using

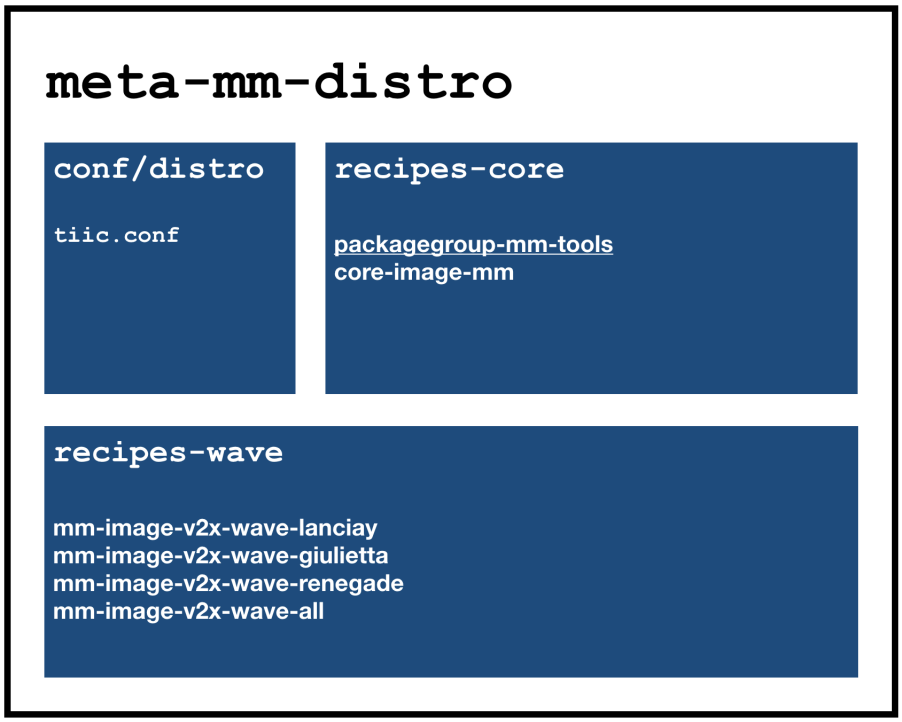


Figure 5.3: The recipes arrangement in the distro `meta-mm-distro` layer.

the Yocto Project facilities, the **eSDK** should be generated and distributed. In particular, the workflow described in 4.6.5 is modified in the following way:

- Identification of the required libraries and dependencies of the software module from the **OpenEmbedded Layer Index**.
- Creation of a new module for containing the software component on the Framework **Subversion** repository.
- Generation of the software unit **recipe** by using `devtool add` and fine-tuning of the recipe via `devtool edit-recipe`.
- Iteration of the following phases until the development is complete:
  - Coding phase with an IDE chosen by the devleoper.
  - Compilation of the software unit via `devtool build`.
  - Automatic **deploy** of the unit and all the required libraries via `devtool deploy-target`.
  - Debugging via `gdb`.
  - Automatic **undeploy** of the unit with all of its dependencies via `devtool undeploy-target`.

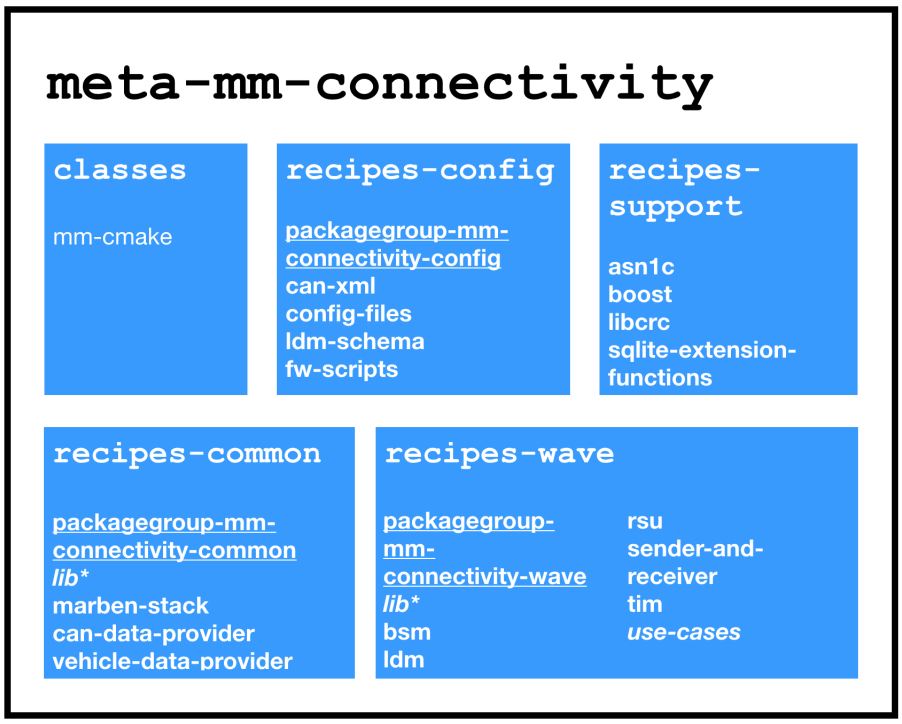


Figure 5.4: The recipes arrangement in the application `meta-mm-connectivity` layer.

  - Commit of the generated code on the `trunk` area of the Subversion module.

- Creation of a `tag` in the Subversion module containing the finished software unit.

- Commit of the generated recipe in the `meta-mm-connectivity` repository

- On **the Yocto Project machine**, integration of the software module in the **TIIC distro images**.

5.6 Issues and solutions

During the development and testing phases of the `meta-mm-*` layers and the TIIC distribution several issues were encountered. Most of all the problems were caused by the inherent custom nature of the Step 03 board, while other problems were caused by the Magneti Marelli corporate firewall.

5.6.1 The Step 03 kernel

As already cited, the biggest difficulty encountered with the Step 03 was to find a working Linux kernel for it. The approach of this work was **bottom-up**: in a sense that I started from testing an almost **vanilla Freescale Linux** kernel, up to an highly customised kernel for the board from which the Step 03 derives. This approach was taken because the customised kernel contains several references to components and peripherals that are **not used** in the Connectivity context (for example the **audio controller**, the **HDMI** support, etc.). Unfortunately, since the Step 03 shares the majority of its components with its ancestor board, the only Linux kernel working without further modifications was the aforementioned custom one.

5.6.2 Git repositories and cloud computing

Another issue faced during development phases of the MM Connectivity Yocto Project support, was the difficulty of accessing **external Git repositories** from inside Magneti Marelli. This is prevented by the company firewall for security reasons, but that makes almost impossible to execute the BitBake **fetch tasks** of the recipes that reference source code stored in Git repositories. The only solution is to fetch all the required repository data by using an external internet connection, and that is in fact the approach that was taken towards the final phases of the work. Anyway, that was unfeasible at the early stage of the work, because several attempts and testing needed to be performed. In order to solve that, I used the **Google Cloud Platform** cloud computing services in order to create a **remote virtual machine** that could temporarily host the Yocto Project build system. This method also significantly **improved build times**, because the technical specifications of the Cloud Platform virtual machines are unquestionably better than my local computer.

Chapter 6

Conclusion

6.1 Results

As seen in the previous chapters, the Yocto Project is a set of extremely powerful tools that can enhance and simplify the development of embedded software and custom Linux distributions. In the context of Magneti Marelli Connectivity the developed solution brings several enhancements, both to the BSP (Board Support Package) team and to the software development team. It will be extremely easy to generate an entire operating system that contains out-of-the-box support to the MM Connectivity Framework, and, thanks to the Yocto Project **eSDK**, the development process will be highly simplified.

6.2 Future developments (Yocto Project context)

Even if the developed `meta-mm` layers provide a substantial number of benefits, several areas of improvement can be identified in order to enhance the possibilities given by the Yocto Project.

6.2.1 Git migration

Git is the de facto VCS (Version Control System) standard for open source development and, especially, for the **Linux kernel**. Git was in fact created specifically for that purpose: facilitating the collaboration in the context of the Linux kernel development. It presents several improvements with respect to other systems like Subversion, and, for that reason, the Yocto Project is built upon the Git principles. Migrating the Magneti Marelli Connectivity source code from Subversion to Git would further improve the development process. `devtool` completely supports Git, so less steps would be needed for committing the generated source code.

6.2.2 Centralised build machine

Thanks to the **shared state cache** mechanism (cited in 3.3.2), the Yocto Project fully supports distributed environments (figure 6.1), both in the **Poky** context and in the **eSDK** context. In the MM Connectivity development environment a server hosting a the shared state cache could dramatically improve build times on developers eSDK machines because all the results of previous compilations could be reused.

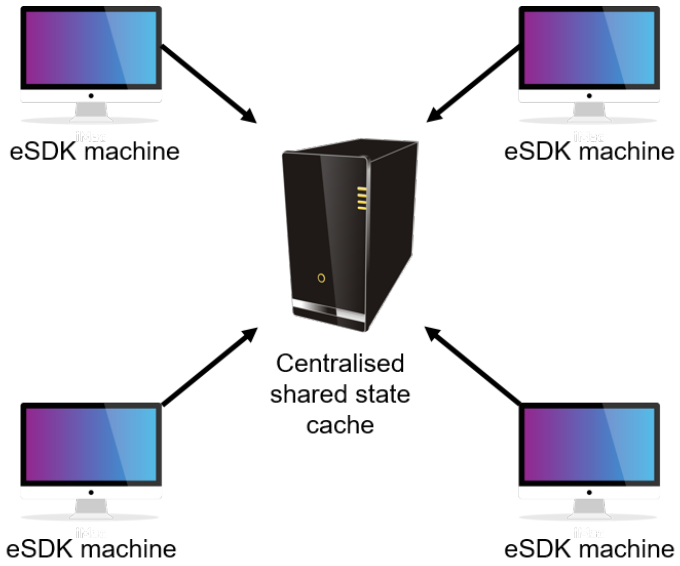


Figure 6.1: Development environment with a server-hosted shared state cache.

6.2.3 Step 03 QEMU support

As cited in 3.4.1, the Yocto Project includes support for the QEMU emulator and virtualizer. By developing proper support for the Magneti Marelli Step 03 board in the QEMU emulator, the software debugging and testing phases could be performed directly on a virtualized environment on the development machine, without deploying the software component on the target board.

6.3 Future developments (Linux kernel context)

During the development of the TIIC distro for MM Connectivity, some areas of improvement have emerged. In particular, the main field in which some enhancements could be performed is the Linux kernel.

6.3.1 Adaptation of the customised Step 03 Linux kernel

As already mentioned in the previous chapters, the Step 03 board design derives from a Magneti Marelli custom-designed infotainment board. The Linux kernel to be deployed on this device was heavily customised in order to obtain proper compatibility with the onboard peripherals. A substantial number of devices needed on this infotainment board are not used for Connectivity purposes, so the Linux kernel could be offloaded from these additions.

6.3.2 Real-time Linux support

Since some of the use cases provided by the Connectivity Framework are strictly safety-related, in certain situations there are some timing constraints to comply with. In a R&D context timeliness can be loosely enforced with some programming constructs, but of course the operating system scheduler would operate independently these choices, thus making them useless. For that reason, another big improvement that could be made to the kernel would be to bring **real-time** support to it. The Yocto Project provides support to the **Real-Time Linux Project** (also called `PREEMPT_RT`), but another suitable alternative could be **RTAI**.

Appendix A

Magneti Marelli Connectivity Yocto Project User Manual

The following appendix completely describes how to **install** the Yocto Project (**Poky**) on a machine, how to **generate** the **TIIC distro** for the **Step 03** board and how to **flash** it.

A.1 Yocto Project installation and usage

This section explains how to install the Yocto Project (**Poky**) and the **Freescale** sets of metadata on the build machine. Furthermore, it is reported how to build the kernel and two `rootfs` flavours for the Magneti Marelli Step 03 board using the Yocto build system. The reference Yocto Project version is 2.4.3 (Rocko).

A.1.1 Requirements

There are some requirements that need to be satisfied in order to have a working build environment:

- Make sure to comply with the system requirements specified by the Yocto Project Quick Start Guide.
- The build machine must be able to use the `git` protocol for fetching the source code from the required repositories. In case this access is not available, the **workaround** specified in A.1.3 should be used.
- Access to the Subversion repository containing the Magneti Marelli Technology & Innovation, Innovation Connectivity layers and the Connectivity Framework source code.

A.1.2 Instructions (with `git`)

By following these steps you can compile the Step 03 Linux kernel **4.1.15** and generate the **UBI** filesystem to be flashed on the board.

1. Install the `repo` tool on your host machine:

   ```
   $ sudo apt-get install -y repo
   ```

2. Create on your host machine the directory `/opt/yocto/`:

   ```
   $ sudo install -o $(id -u) -g $(id -g) -d /opt/yocto
   ```

3. Install the Freescale BSP Yocto release with the following commands:

   ```
   $ mkdir /opt/yocto/fsl-community-bsp
   $ cd /opt/yocto/fsl-community-bsp
   $ repo init -u https://github.com/Freescale/fsl-community-bsp-platform \
       -b rocko
   $ repo sync
   ```

4. Checkout the the `meta-mm` layers in `/opt/fsl-community-bsp/sources/`:

   ```
   $ cd /opt/yocto/fsl-community-bsp/sources
   $ svn checkout <url_to_svn_repo>/meta-mm
   $ svn checkout <url_to_svn_repo>/meta-mm-connectivity
   $ svn checkout <url_to_svn_repo>/meta-mm-distro
   ```

   Check that MM TIIC layers are in the `sources` folder

5. In `/opt/yocto/fsl-community-bsp/` Initialise the build environment:

   ```
   $ MACHINE=imx6qpstep3 DISTRO=tiic source setup-environment build
   ```

6. In `/opt/yocto/fsl-community-bsp/build/conf/local.conf` add the following lines:

   ```
   SVN_USER = "your Subversion username"
   SVN_PSWD = "your Subversion password"
   ```

These two variables are required from the BitBake fetcher for accessing to the TIIC Subversion repository.

7. In `/opt/yocto/fsl-community-bsp/build/conf/bblayers.conf` add the following lines:

```
BBLAYERS = " \
      ${BSPDIR}/sources/meta-openembedded/meta-python \
      ${BSPDIR}/sources/meta-openembedded/meta-networking \
      ${BSPDIR}/sources/meta-mm \
      ${BSPDIR}/sources/meta-mm-distro \
      ${BSPDIR}/sources/meta-mm-connectivity \
...
```

8. To compile the kernel and generate a minimal `rootfs` (containing only basic utilities and kernel modules, **without** the MM Connectivity Framework) issue the following commands:

```
$ cd /opt/yocto/fsl-community-bsp/build
$ bitbake core-image-mm
```

9. To obtain a complete image (containing also the MM Connectivity Framework):

```
$ cd /opt/yocto/fsl-community-bsp/build
$ bitbake mm-image-v2x-wave-all
```

10. The build outputs can be found under the `.../build/tmp/deploy/images/imx6qpstep3/` folder: in particular you will find the **kernel fitImage** (named `fitImage`) and the selected `rootfs` (named `mm-image-v2x-wave-all-imx6qpstep3.ubi` or `core-image-mm-imx6qpstep3.ubi`).

A.1.3 Instructions (without `git`)

In case the build machine is not granted access to external `git` repositories, the following steps should be followed. Please notice that this method is not recommended, because that makes the Yocto Project very **difficult to update** to a new version.

1. Create on your host machine the directory `/opt/yocto/`:

```
$ sudo install -o $(id -u) -g $(id -g) -d /opt/yocto
```

2. Install the Freescale BSP Yocto Project release with the following commands (included in the `sh` script `install_fsl_yocto_rocko.sh`):

```
$ mkdir /opt/yocto/fsl-community-bsp
$ cd /opt/yocto/fsl-community-bsp
$ mkdir sources
$ cd sources
$ wget -P /tmp/yocto/ https://github.com/Freescale
    /Documentation/archive/rocko.zip
$ wget -P /tmp/yocto/ https://github.com/Freescale
    /fsl-community-bsp-base/archive/rocko.zip
$ wget -P /tmp/yocto/ https://github.com/Freescale
    /meta-freescale/archive/rocko.zip
$ wget -P /tmp/yocto/ https://github.com/Freescale
    /meta-freescale-3rdparty/archive/rocko.zip
$ wget -P /tmp/yocto/ https://github.com/Freescale
    /meta-freescale-distro/archive/rocko.zip
$ wget -P /tmp/yocto https://github.com/openembedded/
    /meta-openembedded/archive/rocko.zip
$ unzip "/tmp/yocto/rocko.zip*"
$ mv Documentation-rocko Documentation
$ mv fsl-community-bsp-base-rocko base
$ mv meta-freescale-rocko meta-freescale
$ mv meta-freescale-distro-rocko meta-freescale-distro
$ mv meta-freescale-3rdparty-rocko meta-freescale-3rdparty
$ mv meta-openembedded-rocko meta-openembedded

$ wget -P /tmp/yocto/ http://downloads.yoctoproject.org
    /releases/yocto/yocto-2.4.3/poky-rocko-18.0.3.tar.bz2
$ tar jxf /tmp/yocto/poky-rocko-18.0.3.tar.bz2
$ mv poky-rocko-18.0.3 poky

$ ln -s $PWD/base/setup-environment ../setup-environment
$ ln -s $PWD/base/README ../README

$ rm -rf /tmp/yocto
```

3. Checkout the the `meta-mm` layers in `/opt/fsl-community-bsp/sources/`:

```
$ cd /opt/yocto/fsl-community-bsp/sources
$ svn checkout <url_to_svn_repo>/meta-mm
$ svn checkout <url_to_svn_repo>/meta-mm-connectivity
$ svn checkout <url_to_svn_repo>/meta-mm-distro
```

Check that MM TIIC layers are in the `sources` folder

4. In `/opt/yocto/fsl-community-bsp/` Initialise the build environment:

   ```
   $ MACHINE=imx6qpstep3 DISTRO=tiic source setup-environment build
   ```

5. In `/opt/yocto/fsl-community-bsp/build/conf/local.conf` add the following lines:

   ```
   SVN_USER = "your Subversion username"
   SVN_PSWD = "your Subversion password"
   ```

   These two variables are required from the BitBake fetcher for accessing to the TIIC Subversion repository.

6. In `/opt/yocto/fsl-community-bsp/build/conf/bblayers.conf` add the following lines:

   ```
   BBLAYERS = " \
         ${BSPDIR}/sources/meta-openembedded/meta-python \
         ${BSPDIR}/sources/meta-openembedded/meta-networking \
         ${BSPDIR}/sources/meta-mm \
         ${BSPDIR}/sources/meta-mm-distro \
         ${BSPDIR}/sources/meta-mm-connectivity \
   ...
   ```

7. At this point you have to copy an already populated `downloads` folder in `/opt/yocto/fsl-community-bsp/`. That is needed because several recipes of the Poky and Freescale metadata use `git` as a fetcher. By putting the `downloads` folder already containing the required files, BitBake can skip the `git` fetching tasks.

8. To compile the kernel and generate a minimal `rootfs` (containing only basic utilities and kernel modules, **without** the MM Connectivity Framework) issue the following commands:

```
$ cd /opt/yocto/fsl-community-bsp/build
$ bitbake core-image-mm
```

9. To obtain a complete image (containing also the MM Connectivity Framework):

```
$ cd /opt/yocto/fsl-community-bsp/build
$ bitbake mm-image-v2x-wave-all
```

10. The build outputs can be found under the `.../build/tmp/deploy/images/imx6qpstep3/` folder: in particular you will find the **kernel fitImage** (named `fitImage`) and the selected `rootfs` (named `mm-image-v2x-wave-all-imx6qpstep3.ubi` or `core-image-mm-imx6qpstep3.ubi`).

A.2 Flashing the TIIC distro on the Step 03

Since the Step 03 board is equipped with an onboard flash memory, for installing the generated TIIC distro on it some **flashing** operations are needed.

A.2.1 Requirements

In order to perform the flashing operations the following tools are needed:

- A computer with a serial console like PuTTY.
- A pen drive with a FAT32 partition containing a working Linux `fitImage` and a `rootfs` partition.
- On the Linux `rootfs` on the pen drive, the `mtd-utils` should be available.
- The Step 03 NAND memory should be already partitioned with 3 MTD partitions (bootloader, kernel and `rootfs`) in the following way:

```
device nand0 <gpmi-nand>, # parts = 3
 #: name            size            offset          mask_flags
 0: boot            0x00560000      0x00000000      0
 1: kernel-appl     0x00800000      0x00560000      0
 2: ev_fs           0x1ea00000      0x00d60000      0
```

A.2.2 Instructions

Those are the steps to follow for installing the TIIC distro on the Step 03:

1. Connect the board to a computer via serial communication, and start the serial console.
2. Copy the `fitImage` and the UBI `rootfs` volume on the FAT32 pen drive partition.
3. Connect the pen drive via USB OTG to the board.
4. Turn on the board and drop to the U-Boot bootloader console.
5. With the following U-Boot commands erase the UBI kernel partition and flash the Linux kernel `fitImage` in it (**please notice that these commands are valid only with the Step 03 specific UBI partitions**):

   ```
   $ usb start; fatload usb 0:1 0x12000000 fitImage;
   $ nand erase 0x00560000 0x00800000
   $ nand write 0x12000000 0x00560000 0x00800000
   ```

6. Erase the `rootfs` with the following command:

   ```
   $ flash_erase /dev/mtd2 0x00d60000 0x1ea00000
   ```

7. Boot from the Linux kernel `fitImage` loaded on the pen drive (for example called `mykernel`):

   ```
   $ setenv mybootargs setenv bootargs console=ttymxc3,115200
     noinitrd root=/dev/sda2 rootwait rw
   $ usb start; fatload usb 0:1 0x12000000 mykernel
   $ run mybootargs; bootm 0x12000000#conf@1
   ```

8. On the booted Linux mount the FAT32 partition containing the `.ubi` of the TIIC `rootfs`.
9. Flash the UBI `rootfs` (for example called `image.ubi`):

   ```
   $ ubiformat /dev/mtd2 -f image.ubi
   ```

Appendix B

eSDK User Manual

B.1 User manual for building the eSDK

The following manual explains how to generate a Yocto Project **eSDK Installer** starting from a complete installation of the YP.

B.1.1 Instructions

1. Follow the instructions reported in A.1 and make sure that the `mm-image-v2x-wave-all` image is built without problems.
2. With the Build Environment set (with the `setup-environment` script) execute:

   ```
   $ bitbake mm-image-v2x-wave-all -c populate_sdk_ext
   ```

   The eSDK installer should be located in `/opt/yocto/fsl-community-bsp/build/tmp/deploy/sdk`.
3. Distribute the `sdk` folder to the developers.
4. The eSDK can be installed by executing the `sh` script contained in the `sdk` folder.
5. Specify the target folder for installing the eSDK.

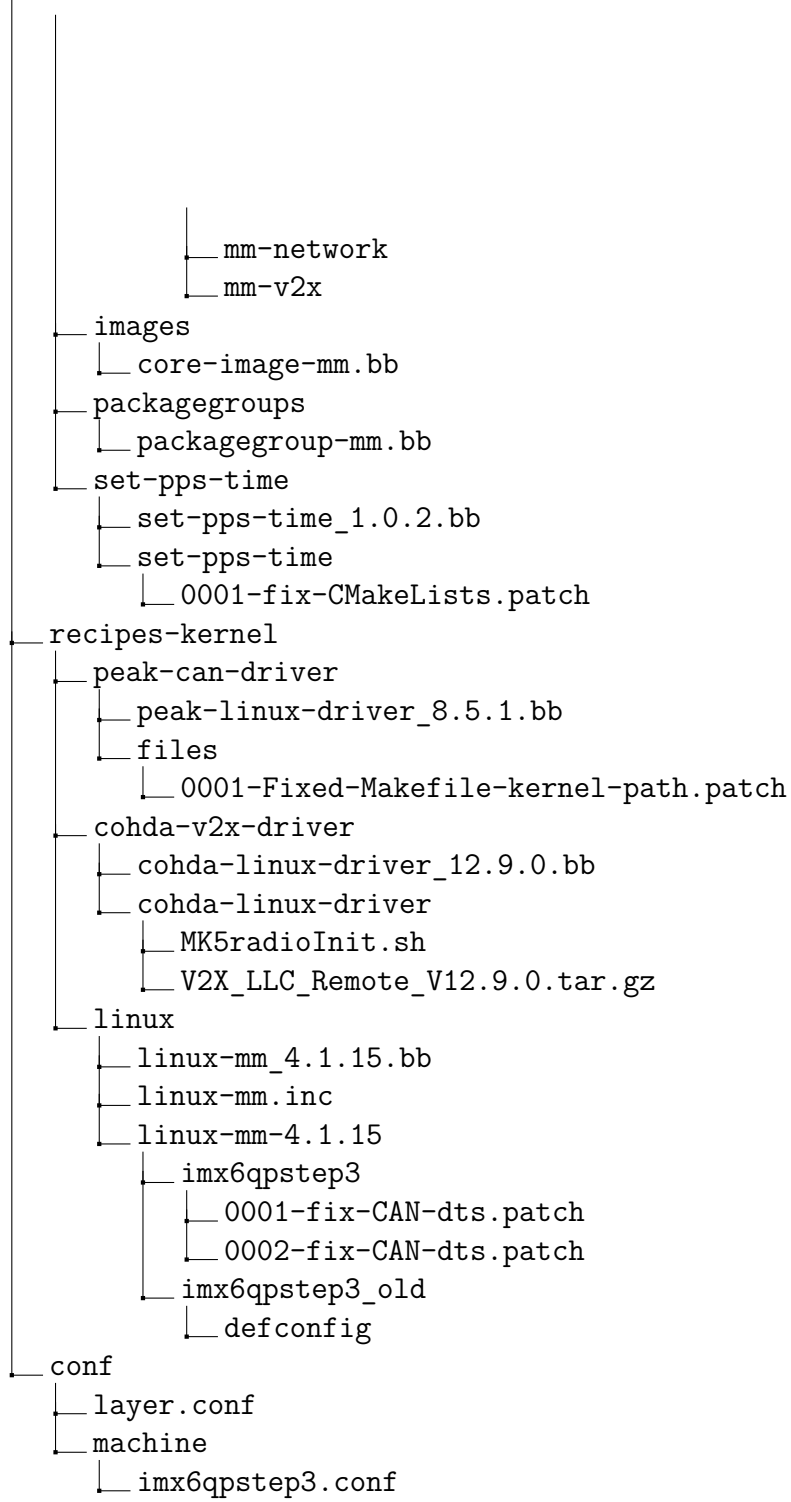
Appendix C

MM Connectivity Layers Directory Trees

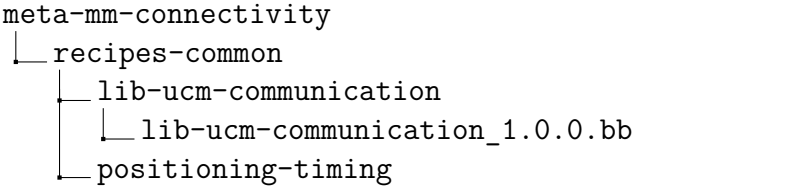
The following directory trees represent the structure of the metadata layers developed for the Magneti Marelli Connectivity environment. It must be noticed that all the patches made to the `CMakeLists.txt` (cited in 5.3.2) are omitted from these trees in order to not overcrowd the diagrams.

C.1 `meta-mm`

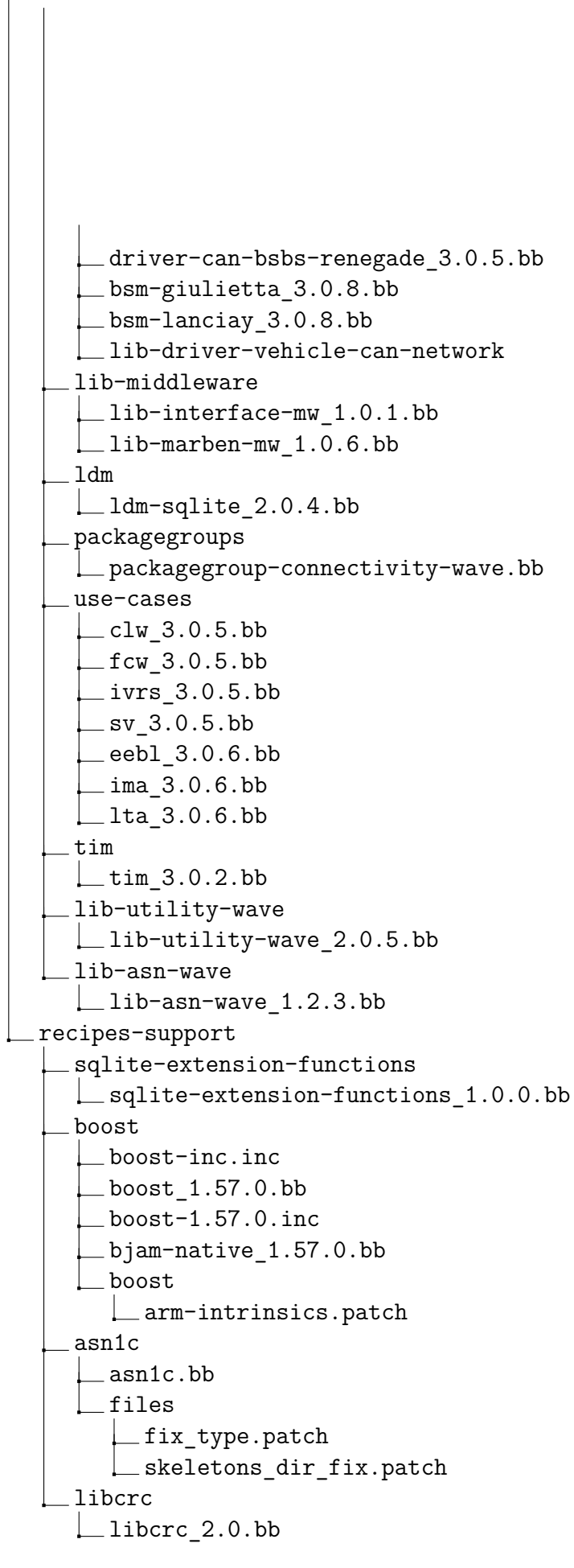
```
meta-mm
├── README
├── recipes-navigation
│   └── gpsd
│       ├── gpsd-machine-conf_%.bbappend
│       └── gpsd-machine-conf
│           └── imx6qpstep3
│               └── gpsd-machine
└── recipes-core
    ├── udev-rules-step3
    │   ├── udev-rules-step3.bb
    │   └── udev-rules-step3
    │       ├── 11-step3-serial.rules
    │       └── 45-pcan.rules
    └── init-scripts
        ├── init-scripts.bb
        └── init-scripts
            └── imx6qpstep3
                ├── mmGpsUpdateSystemTimeOneShot.py
                └── defboard
```



C.2 meta-mm-connectivity



```
│   ├── poti-mediator_1.0.5.bb
│   └── gps-data-provider_1.0.5.bb
├── marben-stack
│   ├── marben-stack_1.3.1.bb
│   ├── lib-v2x-marben-nosec_4.8.bb
│   └── marben-stack
│       └── 0001-fix-Makefile.patch
├── hmi-messages-dispatcher
│   └── hmi-messages-dispatcher_1.0.0.bb
├── uc-manager
│   ├── ucm-tester_3.0.9.bb
│   └── uc-manager_3.0.9.bb
├── lib-ipc
│   └── lib-ipc_1.0.5.bb
├── lib-logger-benchmarking
│   └── lib-logger-benchmarking_1.1.4.bb
├── lib-can
│   ├── lib-can-interface-genivi_2.2.4.bb
│   ├── lib-can-interface_2.1.3.bb
│   └── lib-log-can_1.0.3.bb
├── packagegroups
│   └── packagegroup-connectivity-common.bb
├── lib-pps-ticker
│   └── lib-pps-ticker_1.0.0.bb
├── lib-utility
│   └── lib-utility_3.0.6.bb
└── vehicle-data-provider
    ├── can-data-provider_1.0.5.bb
    └── vdp-mediator_1.0.7.bb
recipes-wave
├── sender-and-receiver
│   └── sender-and-receiver-wave_2.0.3.bb
├── rsu
│   └── rsu-backend_1.0.6.bb
└── bsm
    ├── driver-can-bsbs-giulietta_3.0.5.bb
    ├── bsm-fca_3.0.8.bb
    ├── bsm_3.0.8.inc
    ├── lib-interface-driver-can-network_3.0.4.bb
    ├── driver-can-bsbs-lanciay_3.0.5.bb
    ├── bsm-renegade_3.0.8.bb
    └── driver-can-bsbs-fca_3.0.5.bb
```


driver-can-bsbs-renegade_3.0.5.bb
bsm-giulietta_3.0.8.bb
bsm-lanciay_3.0.8.bb
lib-driver-vehicle-can-network
lib-middleware
lib-interface-mw_1.0.1.bb
lib-marben-mw_1.0.6.bb
ldm
ldm-sqlite_2.0.4.bb
packagegroups
packagegroup-connectivity-wave.bb
use-cases
clw_3.0.5.bb
fcw_3.0.5.bb
ivrs_3.0.5.bb
sv_3.0.5.bb
eebl_3.0.6.bb
ima_3.0.6.bb
lta_3.0.6.bb
tim
tim_3.0.2.bb
lib-utility-wave
lib-utility-wave_2.0.5.bb
lib-asn-wave
lib-asn-wave_1.2.3.bb
recipes-support
sqlite-extension-functions
sqlite-extension-functions_1.0.0.bb
boost
boost-inc.inc
boost_1.57.0.bb
boost-1.57.0.inc
bjam-native_1.57.0.bb
boost
arm-intrinsics.patch
asn1c
asn1c.bb
files
fix_type.patch
skeletons_dir_fix.patch
libcrc
libcrc_2.0.bb

```
        ├── libcrc
        │   └── CMakeLists.txt
    ├── conf
    │   └── layer.conf
    ├── recipes-config
    │   ├── ldm-schema
    │   │   └── ldm-schema_1.0.0.bb
    │   ├── config-files
    │   │   └── config-files-all_1.1.2.bb
    │   ├── start-scripts
    │   │   ├── start-scripts_1.1.2.bb
    │   │   └── start-scripts
    │   │       ├── stopV2Xfw_MM_WAVE.sh
    │   │       ├── FrameworkStatus.sh
    │   │       ├── startV2Xfw_MM_WAVE.sh
    │   │       ├── startRSU_MM_WAVE.sh
    │   │       └── stopRSU_MM_WAVE.sh
    │   ├── packagegroups
    │   │   └── packagegroup-connectivity-config.bb
    │   └── can-xml
    │       ├── can-xml-renegade_1.0.0.bb
    │       ├── can-xml-lanciay_1.0.0.bb
    │       ├── can-xml-giulietta_1.0.0.bb
    │       └── can-xml.inc
    └── classes
        └── mm-cmake.bbclass
```

C.3 meta-mm-distro

```
meta-mm-distro
├── recipes-wave
│   └── images
│       ├── mm-image-v2x-wave-renegade.inc
│       ├── mm-image-v2x-wave-base.inc
│       ├── mm-image-v2x-wave-giulietta.inc
│       ├── mm-image-v2x-wave-all.bb
│       ├── mm-image-v2x-wave-fca.inc
│       └── mm-image-v2x-wave-lanciay.inc
└── conf
    ├── layer.conf
    └── distro
```

```
└── tiic.conf
```

Bibliography

[1] *BitBake User Manual - Checksums (Signatures)*. URL: https://www.yoctoproject.org/docs/2.4.3/bitbake-user-manual/bitbake-user-manual.html#checksums (visited on 07/23/2018).

[2] Bootlin. *Embedded Linux system development*. URL: http://free-electrons.com/doc/training/embedded-linux/embedded-linux-slides.pdf (visited on 08/27/2018).

[3] H. Bruce. *Yocto Project Extensible SDK: simplifying the workflow for application developers*. 2017. URL: https://events.static.linuxfound.org/sites/events/files/slides/2017%20ELC%20Henry%20Bruce.pdf (visited on 07/22/2018).

[4] *CROPS GitHub Repository*. URL: https://github.com/crops/crops/blob/master/README.md (visited on 07/22/2018).

[5] 1609_WG - Dedicated Short Range Communication Working Group. *IEEE Std 1609.1-2006 - Trial-Use Standard for Wireless Access in Vehicular Environments (WAVE)*. Tech. rep. VT - IEEE Vehicular Technology Society, 2006.

[6] Christopher Hallinan. *Bootloaders in Embedded Linux Systems*. URL: http://www.informit.com/articles/article.aspx?p=1647051 (visited on 08/27/2018).

[7] C. R. Martinez-Eiroa and C. Schmitz. *OpenEmbedded & Bitbake: Open Source Software*. 2012. URL: https://www.denx.de/wiki/pub/ELDKHistory/DocumentationLinks/OpenEmbeddedv1.ppt.

[8] *OpenEmbedded Classic GitHub Repository*. URL: https://github.com/openembedded/openembedded/tree/master/recipes (visited on 10/12/2018).

[9] *OpenEmbedded Manual - Packaging: Defining packages and their contents*. URL: http://www.embeddedlinux.org.cn/OEManual/recipes_packages.html (visited on 07/24/2018).

[10] *OpenEmbedded-Core*. URL: https://www.openembedded.org/wiki/OpenEmbedded-Core (visited on 08/28/2018).

[11] R Purdie. *Yocto Project Architecture Whitepaper*. 2009. URL: https://wiki.yoctoproject.org/wiki/Yocto_Architecture.

[12] O. Salvador and Angolini, D. *Embedded Linux Development using Yocto Projects - Second Edition: Learn to leverage the power of Yocto Project to build efficient Linux-based products*. Packt, 2017. ISBN: 9781788477833.

[13] *The GNU C++ Library Manual - Dual ABI*. URL: https://gcc.gnu.org/onlinedocs/libstdc++/manual/using_dual_abi.html (visited on 08/02/2018).

[14] *UBIFS - UBI File-System*. URL: http://www.linux-mtd.infradead.org/doc/ubifs.html (visited on 08/30/2018).

[15] Trevor Woerner. "Yocto Project Developer Workflow Tutorial". In: 2015. URL: https://drive.google.com/a/linaro.org/file/d/0B3KGzY5fW7laTDVxUXo3UDRvd2s/view (visited on 07/27/2018).

[16] *Yocto Project Application Development and the Extensible Software Development Kit (eSDK)*. URL: https://www.yoctoproject.org/docs/2.4.3/sdk-manual/sdk-manual.html (visited on 09/01/2018).

[17] *Yocto Project Development Tasks Manual*. URL: https://www.yoctoproject.org/docs/2.4.3/dev-manual/dev-manual.html (visited on 07/12/2018).

[18] *Yocto Project Members*. URL: https://www.yoctoproject.org/ecosystem/members/ (visited on 10/12/2018).

[19] *Yocto Project Overview and Concepts Manual*. URL: https://www.yoctoproject.org/docs/2.5/overview-manual/overview-manual.html (visited on 08/28/2018).

[20] *Yocto Project Software Overview*. URL: https://www.yoctoproject.org/software-overview/ (visited on 07/12/2018).