VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Huong Dang

DEVELOPMENT OF LINUX DISTRIBUTION USING YOCTO PROJECT

Unit Technology and Communication
2022

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology Bachelor

ABSTRACT

Author	Dang Mai Huong
Title	Development of Linux Distribution Using Yocto Project
Year	2022
Language	English
Pages	43
Name of Supervisor	Jukka Matila

The purpose of this thesis was to develop an Embedded Linux distribution for a customed Arm-based board named KK2040 of Procemex Company. This aim was achieved by using the Yocto Project set of tools.

During this thesis, the Yocto project technologies as well as the Linux platform were studied from the point of view how to integrate software development and update it to the device hardware.

All these outputs are represented by a camera image which contains the built Yocto project and its layers.

Keywords Embedded, system, programming and development

ACKNOWLEDGEMENTS

This thesis is done in Procemex Oy, Tampere from September 2021 to February 2022.

I started my practical training in May 2021 at the company in the role of developing camera and light module software. It was an amazing opportunity for me as a 3rd year student in Vaasa University of Applied Sciences. I learnt so much during the training period and was guided and instructed by my supervisor Kari Mikko and Zalán Rajna.

I knew nothing about the Yocto Project and camera distribution building process before this project and I am thankful for this opportunity granted by the company to expand my knowledge and understand better the Linux Operating System.

I am thankful for my supervisor Mr Markku Selin for being supportive and guiding me through the process and writing this thesis. The courses at Vaasa University of Applied Sciences, especially the Linux Operating System course has helped me a lot to understand the basic structure and developing environment. Lastly, I would like to thank my teacher Jukka Matila for helping me in the process of completing this thesis and being my supervisor.

March 22nd, 2022

Dang Mai Huong

CONTENTS

LIST OF FIGURES AND TABLES

1 INTRODUCTION AND BACKGROUND INFORMATION

1.1 Introduction

An embedded system is a computing device specially created to perform a function, whether to be part of a large system or as an independent device. An embedded system manipulates many electrical and mechanical devices which are widely used. These systems are so unique and application-specific that their development is fundamentally different from general-purpose systems. It is a combination between custom hardware and specifically developed software; therefore, it is important to have proper tools in developing the whole system./1/

The KK2040 camera was developed and created by the Procemex team for its purposes which are captured and analysed pictures. This is an Intel x86 custom board. The camera board is needed to develop its own distribution so that the company can have the full control of the software design and run it with the most efficient rate. Making that an embedded system development project, several challenges needed to be addressed and using the research method for develop, test, and deploy processes were used in this thesis.

1.2 Procemex

The company Procemex is a worldwide pioneer in vision business in the pulp, paper & printing. Procemex is specialized in designing and manufacturing smart camera and lighting solutions for integrated web monitoring and web inspection./2/

1.3 Reasons for Choosing this Thesis Topic

The thesis is about studying a process while rebuilding a custom image from the Yocto project of a camera KK2040 developed by the company. This image was developed and built before by a previous employee and my job was to rebuild the image and update some obsolete information and recipes as well as to consider upgrading the release version of Yocto project of the camera distro (Dunfell

version). Many documents have been written about Yocto, but because this is a custom design, the build is only available in the company and could not be found elsewhere. The image could be rebuilt with the old release version of Yocto, which is Rocko and now in progress to upgrade the image to a Dunfell version. We are alsoe trying to integrate an application called VisionAppster native build to the Dunfell version of the distro.

1.4 Goals

The aim for this thesis was to find ways to develop the camera image with the following achievement and improvement:

- Rebuild the custom embedded Linux kernel successfully
- Up-to-date software and fixing all the obsolete links and files in the current working image
- Automatic compilation of the software to be deployed on the system without manual interaction
- Easy update of libraries and utilities used on the camera
- Updating the camera image to newer version of the Yocto project

All those improvements can be achieved by using useful build systems in the embedded Linux field: the Yocto project. This work was targeted toward guaranteeing support to the custom x86 board.

2 LINUX AND EMBEDDED LINUX

An embedded system is the combination of components which can be broken down into hardware and software, based on a microcontroller or microprocessor that processes digital signals and stores them in memory, controlled by a real-time operating system. Embedded Linux is an embedded system in which the Linux operating system is operating the hardware. /3/

2.1 Embedded Linux

Since 2000, the Linux kernel is increasingly used in the context of embedded system. Linux is the perfect candidate for embedded system with several advantages from developer tools to scalability and they are particularly relevant in the embedded environment /4/:

- Open-source and commercial support: The availability of commercial is a great reason to choose Linux for the embedded system. Because Linux is widely used, and the source code is freely online, helping developers to access and get help from a large community from other developers. The community-maintained systems, such as Yocto and Buildroot enable users to create custom Linux distros for their own purposes.
- High quality but low cost: Many operating systems are expensive and require royalties to be paid. The Linux operating system allows for multiple suppliers of software, support and development while having a stable kernel with facilitates the ability to read, modify and redistribute the source code.
- Customizability: The Linux operating system is completely customizable. If needed, users can choose the features and then implement them into existing distribution or create their own distribution as a desired version of Linux.

The Linux embedded system has several key components which are: kernel, boot-loader, root filesystem, services, and applications/programs.

2.1.1 The Bootloader and Linux Kernel

When the device is powered on, it will perform initial setup steps and then load a bootloader into the memory and run the code. The bootloader needs to find the right binary program of the operating system and then load that Linux kernel and run the operating system /5/. The boot and startup process follows the sections shown in **Figure 1**.

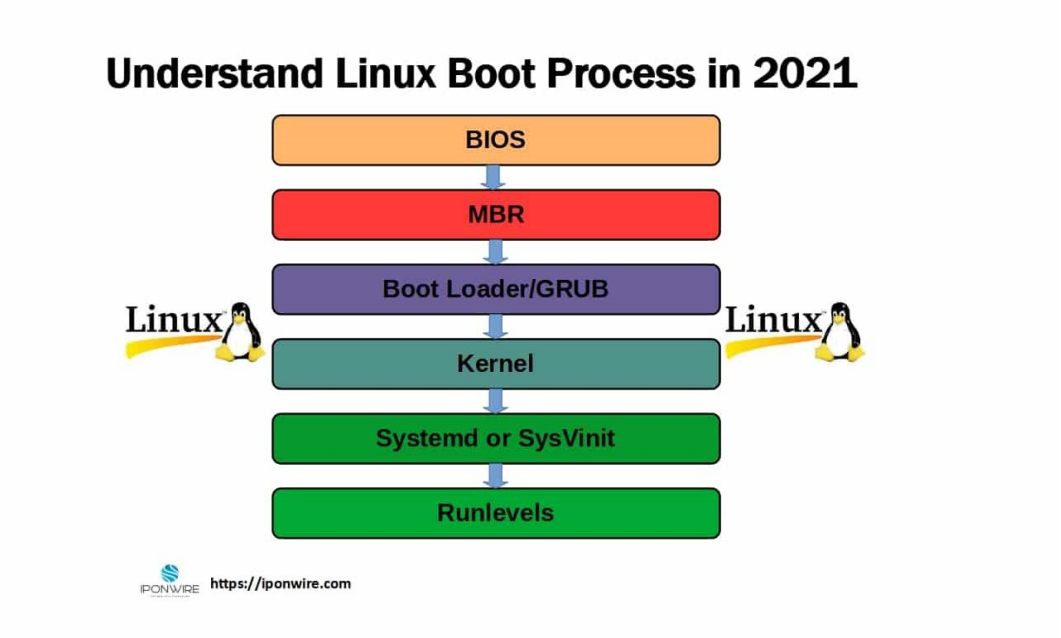

Figure 1. Linux Boot Process /6/

The kernel starts the code which will initialize the hardware, system critical data structures, the schedulers and all the hardware drivers, as well as the filesystem drivers, among other things.

The Linux kernel is one of the most successful open-sourced projects ever created. The difference in the Linux kernel compared to an embedded system is that it is created to run on a different CPU architecture. /29/ The Linux kernel is monolithic and preemptive. The main features of the Linux kernel are:

- Portability: The code can be modified and compiled on different devices.
- Scalability: Possible to be processed from embedded devices to supercomputers.
- Modularity: The kernel just needs to include what it needs.

Linux provides a complete configuration mechanism which is needed to minimize the size of kernel compilation. With the Linux kernel source code, developers can configure which options are decided as needed for the device: Architecture specification, device drivers, filesystem drivers and network protocol. The kernel configuration is in ".config" file which is in the root directory of the kernel source code shown in

Figure 2: "/usr/src/linux/".

The kernel compilation process can start when the kernel configuration has been achieved. The output of this process is a set of files which result in a kernel image and others required from configuration files.

The kernel image is a single file which has satisfied all options in the configuration file, and it is needed to be loaded from the bootloader when powering on the device.

Kernel headers are components which help to compile drivers and the kernel modules which link to the kernel. The developers do not need to install the full sources and thus save some space.

Kernel modules are the part of code that can be loaded into the kernel, depending on the requirement so that the system does not need to be rebooted. Kernel modules can be loaded or unloaded dynamically after the filesystem is mounted to the kernel.

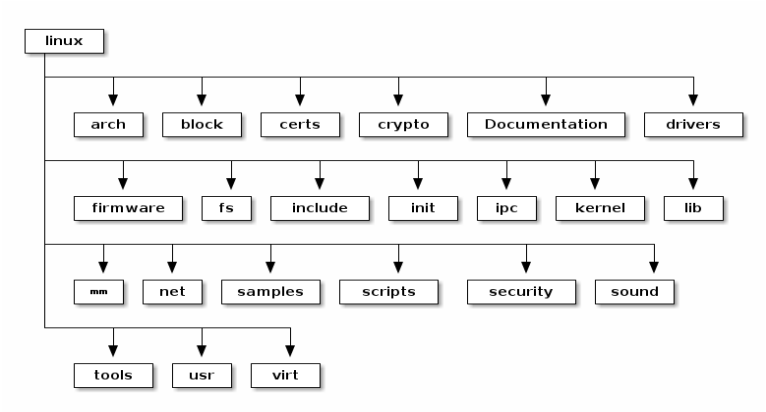

Figure 2. Linux source code layout /7/

There are already some default configuration options but developers still can create new ones. It is advised to not edit the file manually but with a “make menuconfig” command to take care of all the dependencies of each option. Saving changes in the “menuconfig” will update the content of the “.config” file.

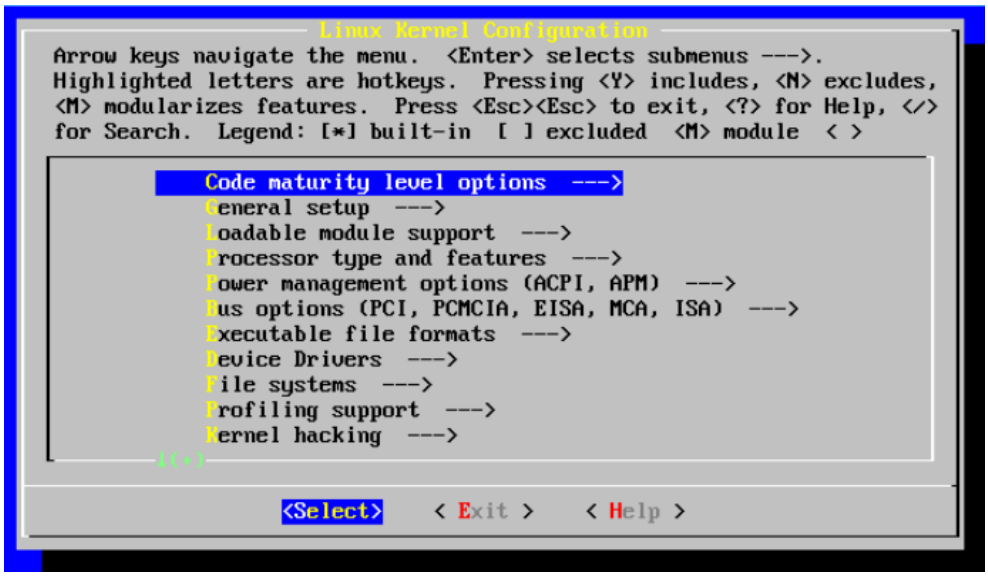

Figure 3. "Make menuconfig" configuration

2.1.2 The Root Filesystem

The root file system, also named rootfs, is one of the key components in Linux. It is the top-level directory of the filesystem that includes all the files which are required, such as libraries and executables to boot the Linux system.

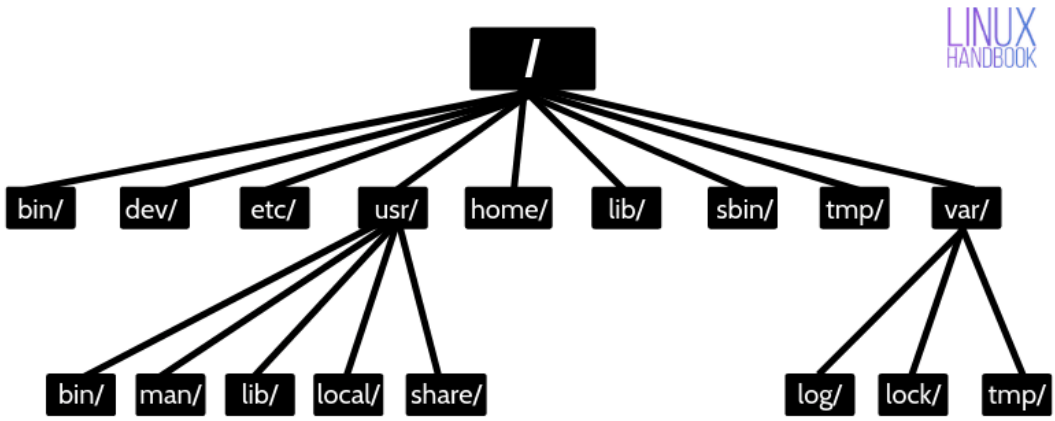

Figure 4. Linux directory structure /8/

Libraries are extremely important will be included in rootfs (normally under the/lib and /lib64 directories), when other non-basic libraries can be found in the /usr/lib directory.

Furthermore, some basic applications, such as command shell or basic system binaries, such as the mount command or non-basic binaries, can be found in rootfs.

BusyBox is a software suite that provides a binary containing needed application. These applications have needed utilities that help Linux operating system to function properly.

2.1.3 Init Manager

The purpose of the Init manager is to start services during boot time. These services could require multiple dependencies and most of the services are daemon. One of the most important Linux initialized systems and service managers are systemd and SysVinit. It provides a logging daemon and utilities which are needed to help the system administration tasks.

2.1.4 Device Manager

The device manager is a component to examine the details of the device hardware and makes them seen the in /dev directory. The device drivers will recognize the peripherals, then mount the devices under the same /dev directory.

2.2 U-BOOT Bootloader

The most famous boot loader in Linux embedded system devices is U-Boot. It is an open-source bootloader with wide range support of microprocessors, such as ARM, and x86 or FPGA based platform. U-Boot also provides a command line interfaces for developers to manipulate, for example, flash memory partitions, load files, and set environment variables. .

2.3 Development Tools

Several development tools are needed to develop the software of Embedded Linux.

2.3.1 Cross-compiling Toolchains

The cross-compiler is used for an architecture different from the host machine which is attached to the Binutils (set of tools for generating and manipulating binaries for given CPU architecture), the kernel headers, libraries and debuggers that are called a cross-toolchain.

The Application Binary Interface is in between two software components, which will define calling conventions, data alignment and data types. This interface must be declared when build toolchain.

When compiling a binary, the compiler must include debug symbols to make debugging possible by 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 stripping operation is executed to remove the debug information from files.

2.3.2 Build Process Management

For a large project, it is not convenient to manually compile and link each component, and it is difficult to write by hand a Makefile for each unit. Therefore, some utilities like CMake and autotools are employed.

CMake helps to manage the building, testing, and packaging process of software development in a large project. It will generate a Makefile with “make”. CMake parses a CMakeLists.txt file that can be modified by the user. CMake supports directory hierarchies, dependencies from several libraries and can be used with cross-compilation.

Autotools are also known as GNU Build System which is a set of tools that allow portability of software on different UNIX systems. Autotools generate a Makefile to be used by “make” and autotools consist of Autoconf, Automake, and Libtool that are needed to build a project.

3 THE YOCTO PROJECT

3.1 Definition

In general, the Yocto project is an open-source set of tools to create a custom embedded Linux distribution. The main purpose of Yocto is to build a Linux-based embedded system which has a platform to support developers from a dedicated community and its members. The Yocto project offers an open-source embedded Linux Yocto build system with package metadata and an SDK generator.

The Yocto project will help developers to custom a desired Linux embedded distribution by producing the Linux kernel with root filesystem containing needed utilities, libraries, and applications. Each task that is created by Yocto Project is executed with a fresh build environment that follows Yocto Project rules.

3.2 Advantages and Disadvantages and Needed Development Environment

3.2.1 Benefits of Using Yocto Project

The Yocto Project supports different architectures in a one build tree. This build tree includes many layers that help flexibility in developing. The Yocto Project layers architecture is a neat way to handle the filesystem and other parts of the system and developers can focus on each layer at its own pace. Using priority ordering, layers can be included and allow higher priority layers to override and modify settings.

Because the Yocto Project has a wide range of users with community support, it is much easier to get the constant support from other developers. As an open-source project, developers can choose and modify the layers with no concern about software vendor changing strategies.

The Yocto Project has a release schedule that is followed strictly. This helps developers to predict and allow development in the right time for projects that are

based on Yocto and to create newest branches which include the latest features for the latest version.

The Yocto Project has recipes to build a functional toolchain[1] that is required in the device system and these toolchains are used by the Yocto community, so they are always tested and coming up with the patch to fix the problems. For application developers, tools and libraries can also be bundled by the Yocto project to make a Software Development Kit SDK, which then gives developers a good overview of all components they need for the project.

3.2.2 Disadvantages of Using the Yocto Project

Unlike the other build systems for a customized Linux embedded system, such as BuildRoot or OpenWrt, the Yocto project uses its own architecture rules with a lot of internal terminology that could be daunting for starters. With a wide range of options to configure the system target, it is confusing to choose the best option for the designed device.

In the Yocto Project, to install new packages, the developers will need to modify and rebuild with no package libraries available on the Internet.

With the cross-build environment, developers need to understand the host and target systems. The developers also need time to run a bitbake build and then deploy the resulted image into the target device. However, after the first deploy, Yocto will offer other intermediate approach to update only needed packages with the build system outputs packages in formats such as rpm, deb, ipk, tar.

3.2.3 Yocto Project Environment Development

The requirements to develop projects in the Yocto Project environment are /9/:

[1] 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.

- A host system with a minimum of 50 GB of free disk space that is running a supported Linux distribution (i.e., recent releases of Fedora, openSUSE, CentOS, Debian, or Ubuntu). If the host system supports multiple cores and threads, the Yocto Project build system can be configured to significantly decrease the time needed to build images.
- Appropriate packages installed on the system that use for builds.
- A release of the Yocto Project.

This Yocto source repository contains IDE Plugins, Matchbox, Poky, Poky Support, Tools, Yocto Linux Kernel, and Yocto Metadata Layers. Developers can find the wanted source to view, access, and create a local copy.

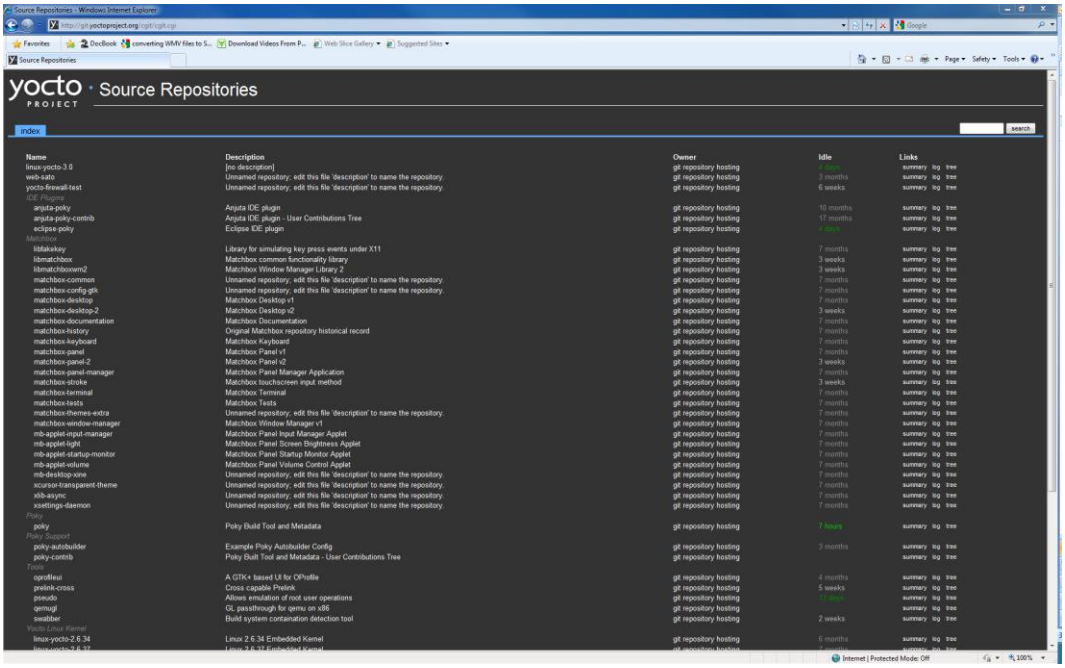

Figure 5. Yocto source repository /10/

3.3 Introduction to the Yocto Project Structure

The core component of the Yocto Project is the Poky Build system. The core component of Poky is BitBake. BitBake is a build tool that uses the Yocto Project and OpenEmbedded community to utilize metadata to create a Linux image from a source. The BitBake executor has several configuration files from OpenEmbedded-

Core named OE-Core. The other main development component of Yocto project is The Yocto project integrated tools. /11/

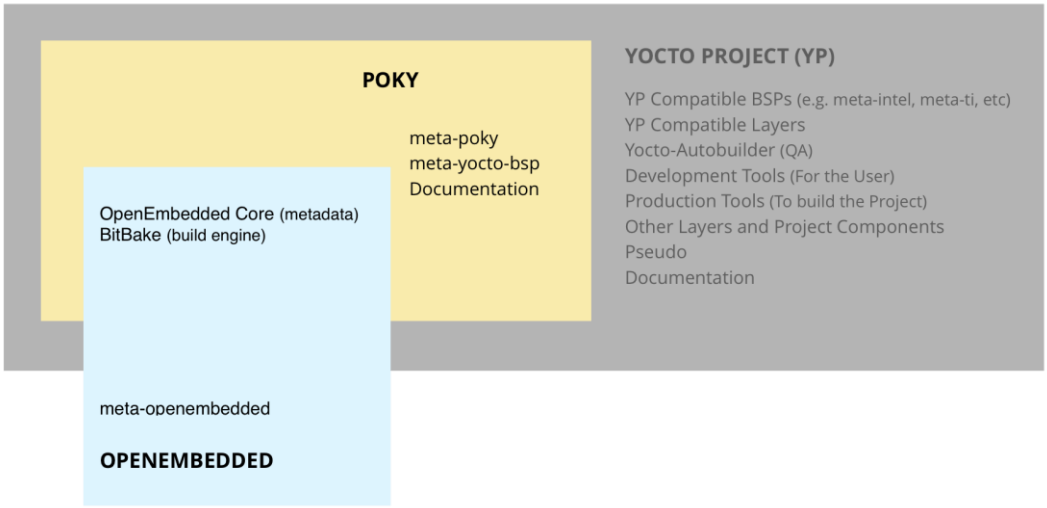

Figure 6. The Yocto Project architecture /12/

After preparing the development environment and designing needed layers (which will be mentioned through section 3.7-3.8), the whole set of tools included can be used for daily basis develop and testing.

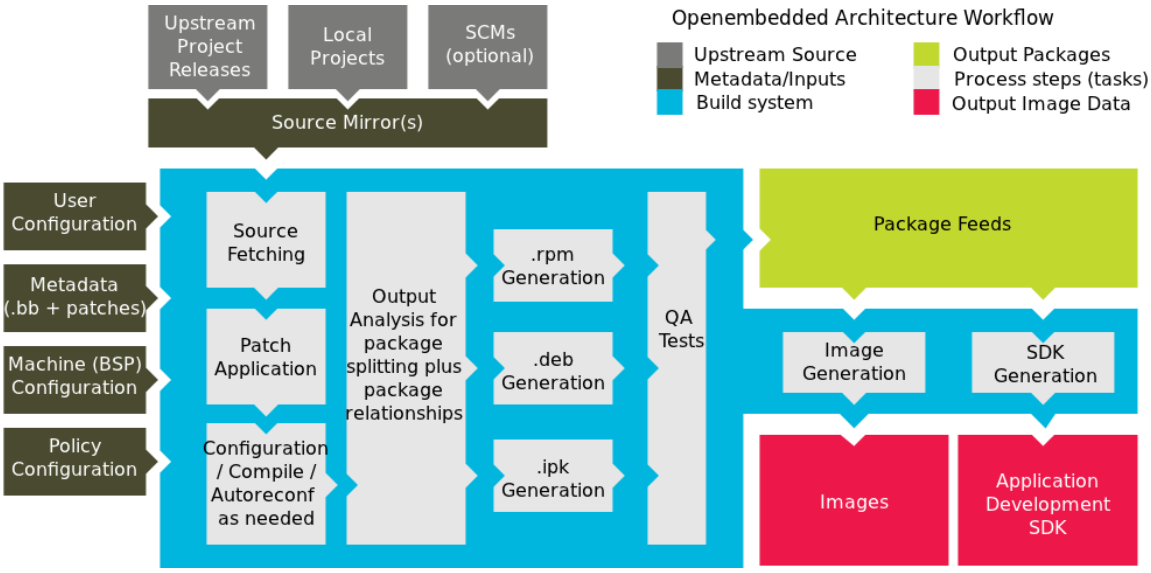

Figure 7. The Yocto project general workflow /13/

3.4 Poky

The Poky build system is a set of files which in default configuration provides an image that can just be an accessible minimal image or a full Linux Standard based image. Metadata layers then can be added to extend the function, or for example, have more software stack, a board support package BSP. However, Poky does not have binary files, it is just an example of how to build a customized Linux distribution from a source. /11/

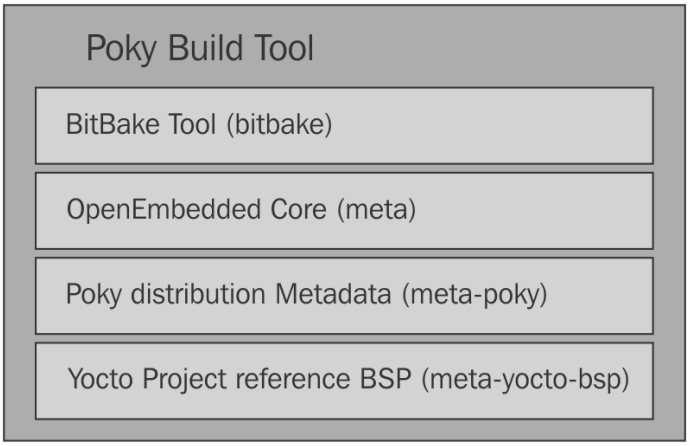

Figure 8. Poky Build Tool /14/

Poky has purposes to provide a minimal functional distro, serve as a test bench for validating and testing the Yocto Project components, and being a self-contained package. The first step to use the Yocto Project is cloning the Poky repository with a suitable version of the Yocto Project using git terminology. Then developers can start testing the building environment with building the minimal image which can be booted to the desired device (this image is called core-image-minimal). This image will include the Poky Linux embedded distribution and an image enclosing a compiled Linux kernel.

On a very high level, the build process starts out by setting up the shell environment for the build run. This is done by sourcing a file, oe-init-build-env, that exists in the root of the Poky source tree. This sets up the shell environment, creates an

initial customizable set of configuration files and wraps the BitBake runtime with a shell script that Poky uses to determine if the minimal system requirements have been met.

3.5 BitBake

BitbBake is the tool of the OpenEmbedded Build system. BitBake parses the Metadata, generates a list of tasks and then executes those tasks. This is called task-oriented approach. Metadata is kept in the recipe which is .bb file and other related recipe appends called ".bbappend" files, configuration files are ".conf" and underlying included files are ".inc" files, and class files which are ".bbclass".

BitBake can fetch libraries from the local directory, remote server, or source control systems, such as a website. The recipe is the instruction of a single piece of software to be built inside the system; these files have information about the unit, such as dependencies, description, checksum, how to compile the source files, how to package the compile output and source for patches if needed.

The BitBake objectives are:

- Provide support for cross-compilation, for tasks within the package recipe (source fetching, patching, configuration)
- Handle complex dependencies
- Be able to interpret tasks written in "sh" or in Python
- Provide checksum (checking mechanism) to improve build speed

Tasks that are executed by BitBake are descripted in two types of metadata which are recipes and configuration files. Metadata is explained in section 3.7

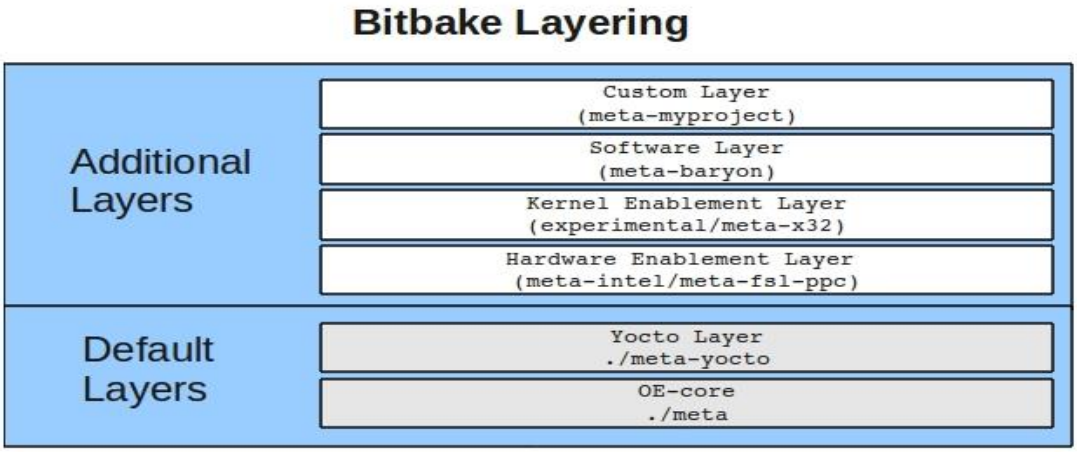

Figure 9. Example of BitBake layering /15/

3.6 OpenEmbedded-Core

OpenEmbedded-Core is a set of metadata, and this is the main layer of the Yocto Project. This layer includes the core recipes for building Linux distro.

OpenEmbedded-Core represents the top foundation having machine, application, and distribution layers. OE-Core is based on a pull model whose patches are sent to the OE mailing list for review, and if it is agreed by the maintainer, it can be merged.

The OpenEmbedded-Core offers to the Yocto Projet the needed recipes and it supports the main architecture 32-bit and 64-bit of ARM, x86,. The out of date version of recipe will be removed from OpenEmbedded-Core; therefore, the recipes will always be the latest version but if the previous one is needed, then that that version must be provided by another layer that are on top of OpenEmbedded-Core.

3.7 The Yocto Project Metadata

The metadata is set of configuration files which are where the Yocto Project processes. These files contain the information of the required software, the command needed to build a component, the patches that are merged by the maintainer.

There is different type of metadata in the Yocto Project:

- Configuration file: includes a definition used by the build system. This file is not only a build instance but also can contain set of metadata (distro and machine /conf files).
- Recipe: files having settings, tasks, package that are required to build the system. There are two types of recipes which are packagegroup and image.
- Layer: used to organize recipes and configuration files

3.7.1 Recipe

A recipe is .bb file that is parsed by BitBake. A recipe will have the information of a piece of software: a source link to obtain the patches that are needed for the software, options for configuration to apply and how to compile the source file and package the compiled output. /30/

A bellow figure is the essential tasks that are executed by the build system and these tasks have its order to be executed.

When creating a new recipe, developers will only need to specify needed tasks, such as “do_install”, “do_compile”, “do_configure” when other tasks will be defined by the Yocto Project build system.

For example, here is a hello-world.bb recipe which is taken from /16/:

```
DESCRIPTION = "Simple helloworld application"

LICENSE = "MIT"

LIC_FILES_CHKSUM = "file://${COMMON_LI-
CENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"

SRC_URI = "file://myhello.c"

S = "${WORKDIR}"

do_compile() {

${CC} myhello.c ${LDFLAGS} -o myhello

}

do_install() {

install -d ${D}${bindir}

install -m 0755 myhello ${D}${bindir}

}
```

To fetch and unpack the sources for the recipe, the do_fetch and do_unpack is needed to complete the jobs. For a local file, it will need the SRC_URI statement in the above code to insert the file then do_fetch and task and all tasks depending on it will re-executed. /17/

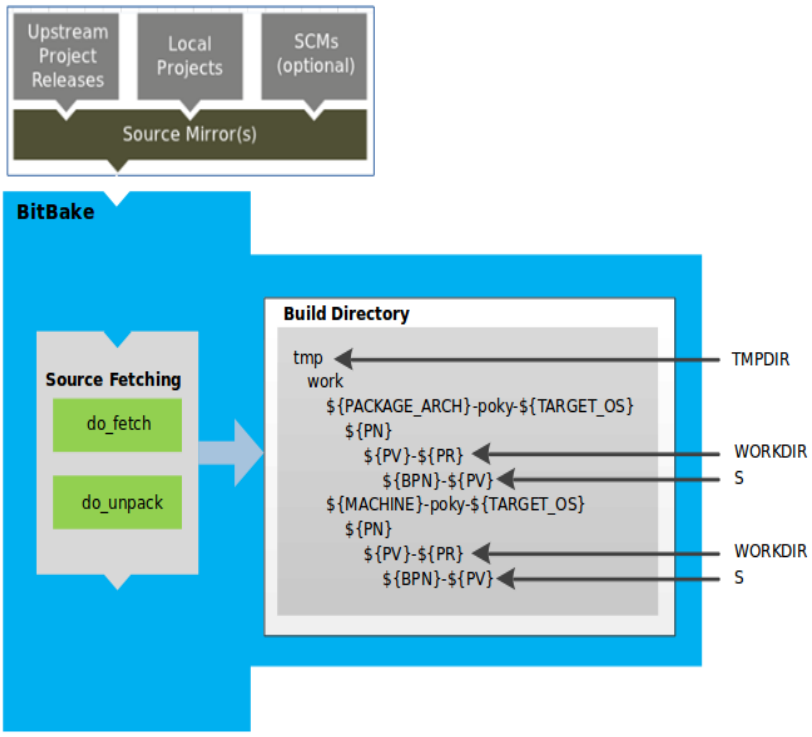

Figure 10. Fetch sources and unpack thems by do_unpack and do_fetch /13/

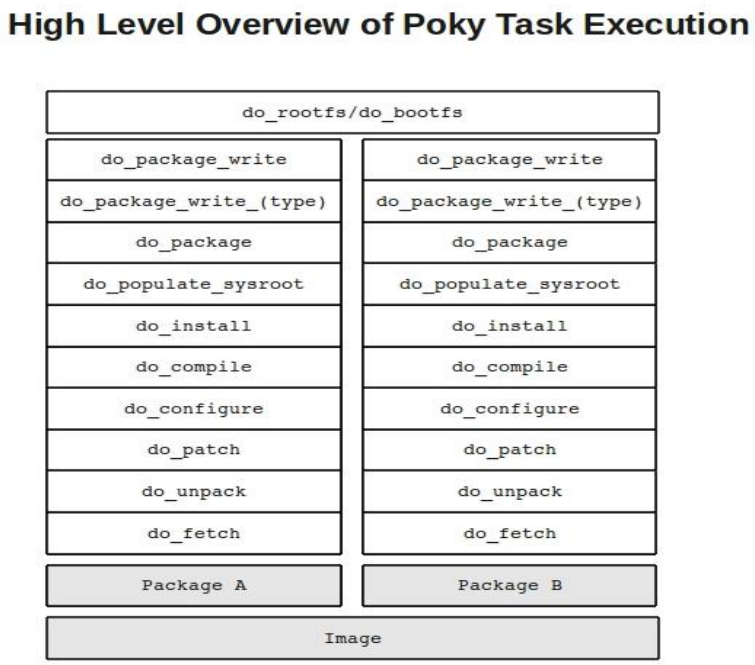

Figure 11. High-level overview of Poky task execution /15/

The Image recipe is the top-level which is a BitBake recipe that contains all the packages. There are some basic Yocto images which can be temporarily inherited for defining new images /18/:

- core-image-minimal: a bootable small image that only has the basic required components for booting this image to the targeted device. This image does not include kernel-modules.

- core-image-base: a full console-only image that includes the board kernel modules

- core-image-sato: an image supporting the Sato environment which is suitable for mobile devices.

Package groups are needed to keep track of all the packages that are required for the image. Therefore, the class recipe named packagegroup.bbclass will be in charge of package groups and perform correct generation of packages. /19/

3.7.2 Configuration File

Configuration files (.conf) contain the configuration variables that govern what Poky does. There are many .conf files that are in different layers of the Yocto Project. /20/

The first step to build the image is to execute the initial script, which is oe-init-dev, and in this executing process, configuration files are built . These files are needed for the starting build process:

- local.conf: a local configuration file which contains current build environment information. Important variables need to be set in this file such as: MACHINE (target hardware device), DISTRO (what distro will be used for the target device), IMAGE_INSTALL_append (install packages in the resulted image).

- bblayers.conf: listing layers that will be checked during the recipe parsing process by BitBake.

There is also the distro .conf file which will contain specific information of the build configuration policies, details of the SDK, version of recipes and packages. All the important variables will be described when cloning the Poky from the Yocto Project source code.

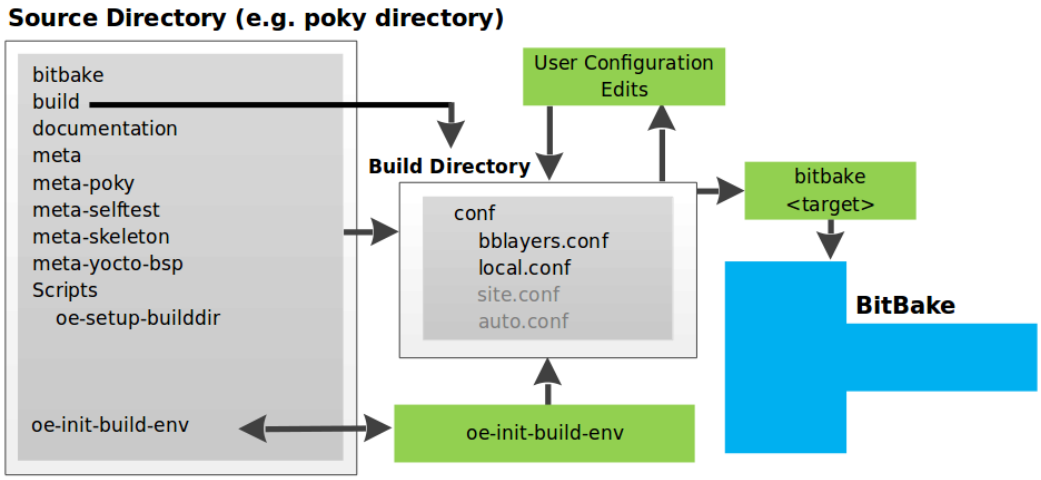

Figure 12. User configuration is presented as this flow /13/

3.7.3 Layers

Using a layer is one of the improvements from the classic OpenEmbedded project. Each layer is different and separated from each other but in general, all layers have the similar structure which will be showed in Figure 13.

The naming of Yocto Project layers is meta-xxx where meta is short for metadata. There are three main types of layers: Distri layers, BSB layers and software layers.

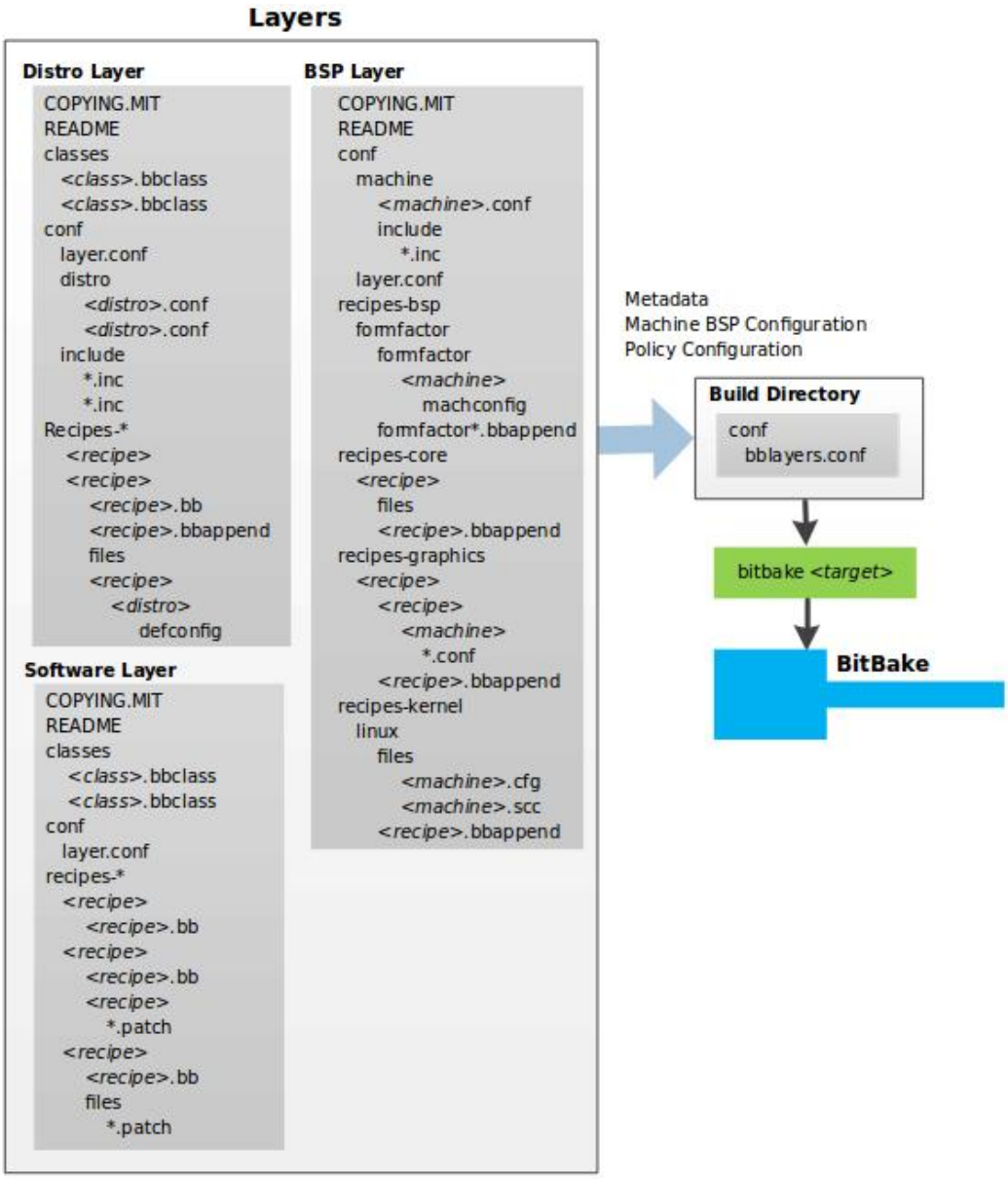

Figure 13. Main types of layers /13/

Each layer will also have a folder conf that include a local configure file for that layer. This file will help BitBake to recognize the layer and its set of metadata.

3.8 The General Yocto Project Workflow

3.8.1 Getting Started

The first step to work with the Yocto Project is to prepare a host machine for running the project. This can be worked on the native Linux operating system, a virtual machine with Linux installation. The build environment will require needed packages and applications to build the image, these required are listed in The Yocto Project Manual.

3.8.2 Working with BSP Layer

Before developing a custom distro for a targeted device, some hardware-specific metadata must be checked with the Yocto Project build. 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 created BSP layer targeted to a single board, first a MACHINE configuration file should be defined. After doing that, the tasks that follow should be performed.

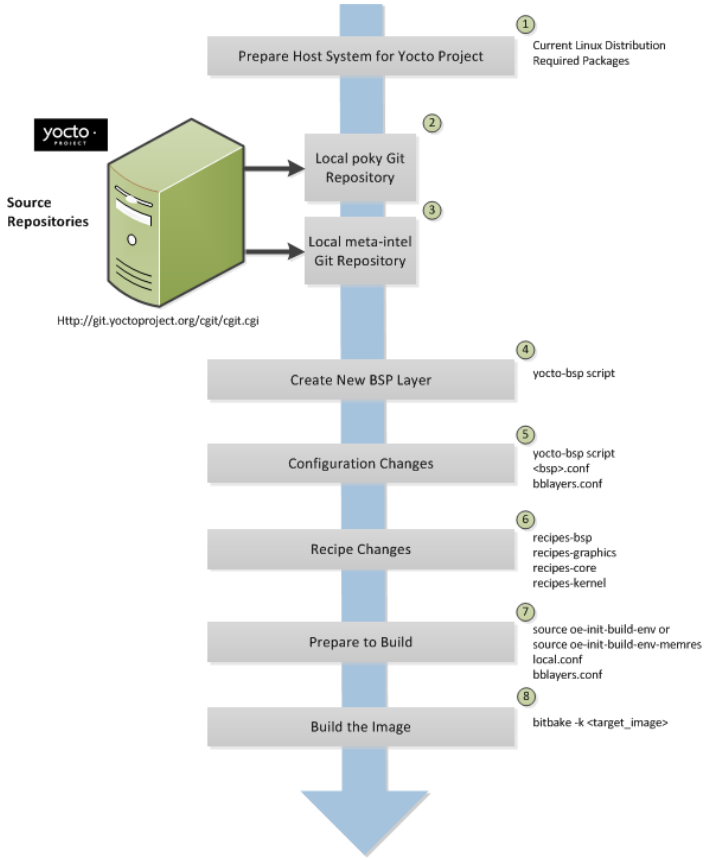

Figure 14. Illustration and list summarize the BSP creation general workflow /21/

Choosing the correct Linux Kernel: 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 a need to create a custom BSP layer, there are various approaches for selecting the right Linux Kernel for an embedded distro. Several vendors (such as Intel and Freescale) provide BSP layers containing kernel recipes pointing to the Linux kernel sources, heavily customised for their architectures that could be reused for the custom embedded system. The first approach would be to define .bbappend files to these Kernel recipes and apply patches to the kernel source via the BitBake do_patch, 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. /22/

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 will produce the .ko files starting from the vendor device driver source code.

3.8.3 Working with Application Layer

As already said, the distro layer is the place where the shape of the embedded Linux distribution is descripted. It contains information about package alternative selections, compile-time options, other low-level configurations and describes the different rootfs images that can be built. /23/

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.

Creating images: Creating an image substantially means to list everything 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 OpenEmbeddedCore layer or could be created by scratch. On both cases, by adding values to the variable IMAGE_INSTALL, packages are included into the output image.

3.9 The Yocto Project with SDK

After finishing the deployment of Yocto Project system, the Yocto Exensible Software Development Kit eSDK will help in process of developing the new software functions. The Yocto eSDK will need to be installed on the development machine. /24/

After building the new recipe, the modification coming from it can be made by the devtool modify command. The devtool tool is handy and saves configuration time.

4 THE YOCTO PROJECT WITH PROCEMEX KK2040 CAMERA

4.1 Introduction

Procemex KK2040 camera already has the board designed and the sensor working inside. The camera software has been developed since 2019 with the Yocto Project to create a customized distro named kuvio-distro. Because the aim of this camera is to process the received images and save the space so that in the future, developers can add more applications or desired functions from customers.

The kuvio-distro is developed with The Yocto Project version Rocko 2.4, and it has its own platform to send the images named kuvio-platform. This thesis will only focus on rebuilding the working kuvio-distro and rebuilding the project with a newer version of the Yocto Project which is the Dunfell version and adding applications to the minimal image.

4.2 Progress of Rebuilding Kuvio-distro

4.2.1 Kuvio-distro and Mender Yocto

A notable point of kuvio-distro is that it is a Mender Yocto Project image. The output image will be able to be flashed to the device storage during initial provisioning and provides an Artifact containing a root filesystem image file that Mender then can deploy to the device.

The meta-mender is a set of layers that will make the Yocto Project image have a part that is a Mender client. With Mender, the image can deploy updates from features such as automatic roll-back, remote management, logging, and reporting. Mender will need to be integrated with the device, most notably with the boot process. The build will be configured the conf/local.conf after running initial steps that will be mentioned in 4.2.2

Mender feature provides a dual rootfs system which allows updating the Linux operating system without overwriting the entire SSD on KK2040. The Linux disk

will have a /data directory which will be preserved over the operating system updates. This means that all the user data (for example, logs, images) should be stored under the /data directory.

Mender supporting distro has three different images: *sda*, *sdb* and *mender*. To install a Linux distro, KK2040 is booted with USB live stick (i.e. /dev/sdb image). It might be necessary to modify the BIOS settings to allow booting from "UEFI mass storage". Once the device is booted from the USB stick and after the user logs on as "root", a script prompts the user to accept installation of the Kuvio distro image to KK2040 SSD drive. Once the user types in 'y' and enters key, the script is executed, and the disk image is copied to the SSD. After the camera is rebooted from SSD (remove the USB stick and modify BIOS settings if needed), another script is automatically executed to extend the initial data partition to use up the rest of the available SSD.

The current configuration of 32GB SSD is roughly the following

- rootfs 1: 1GB
- rootfs 2: 1GB
- boot part: 16MB
- data: 26 GB

4.2.2 Preparing Build Environment for the Yocto Project

The first step is preparing the environment for the building Yocto Project using a Virtual Machine with Ubuntu 18.04 installation as the build host. This build host has a minimum of 50 GB of free disk space that will needed to support the Linux distribution.

From the Yocto Project manual, some packages are required to build an image, which are installed in the host build as the follow command in the terminal:

```
$ sudo dnf install gawk make wget tar bzip2 gzip python3 unzip \
perl patch diffutils diffstat git cpp gcc gcc-c++ glibc-devel \
texinfo chrpath ccache perl-Data-Dumper perl-Text-ParseWords \
perl-Thread-Queue perl-bignum socat python3-pexpect findutils \
which file cpio python python3-pip xz SDL-devel xterm
```

Now the host build has the right packages, this is time to get a copy of the Yocto Project. The release of the Yocto Project used for kuvio-distro v2.4 is named Rocko and when cloning the Yocto repository, the branch should be set to the correct version.

4.2.3 Layers of Kuvio-distro v2.4

The needed repository of The Yoctor Project of kuvio-distro v2.4 is cloned to obtain the script for fetching the right Yocto layers. The poky folder will include these meta layers:

poky
|--meta-intel
|--meta-kuviovision
|--meta-poky
|--meta-qt5
|--meta-raspberrypi
|--meta-xilinx
|--meta-openembedded
|--meta-mender

meta-intel provides carefully chosen tune options and generic hardware support to cover most of the current Intel CPUs. Moreover, the kernel recipe Linux-intel is included, that enhances better Intel hardware support for the Long-Term Support Linux kernel. Because the Procemex KK2040 has an Intel Core i7 CPU, the machine intel-corei7-64 was selected in the local.conf file.
meta-kuviovision is the layer that contains files and libraries needed to support the kuvio platform that run inside the camera as an application for achieving and sending pictures received from the sensor.

Other meta layer is supportive for meta-kuviovision and other purposes that have been mentioned above.

4.2.4 Building /dev/sda and /dev/sdb Image

The sda image file is included in the USB live stick image. So, if the sda image is changed, the new uefiimg file needs to be included in the sdb image so that it gets properly copied to the SSD driver during the installation of the "fresh" KK2040. If the camera already has a mender supporting distro installed, it is enough to introduce the modifications to the distro just by updating .mender image to the camera.

The build process for sdb image is like the sda image. Only the configuration file is different.

After that the next command is used to build the kuvio-distro image:

```
$ bitbake kuvio-image
```

The resulted image will be stored as an uefiimg file in the directory /build/tmp/deploy/images/intel-corei7-64/

To deploy the image into the camera device, a bootable live USB stick is used and the disk image is copied to the USB with the following command:

```
$ dd if=kuvio-distro-intel-corei7-64.uefiimg of=/dev/sdb
```

Assuming that the USB stick is /dev/sdb and it can be checked with “sudo fdisk -l”. After that, the USB stick is inserted to the camera board and break the boot process is broken to enter the BIOS settings. In the settings, the boot device is changed to the USB mass storage and UEFI only option and then continued with the boot process. If the BIOS boot setting was permanently changed, it might be needed to change it back to use the internal SSD.

4.2.5 Build SDK

To build an environment for cross compiling on the host operating system, the following must be executed in the build directory:

```
$ bitbake -c populate_sdk kuvio-image-sdk
```

SDK environment requires a minor tweak to be able to cross-compile Qt software. So, editing the qmake.conf configure file of qt5 is needed.

4.3 Building the Yocto Project with Latest Release – Dunfell

To keep up with the newest version of libraries and packages needed for the camera image, the decision to upgrade the Yocto Project version of image was made. The old version using in the image was released four years ago and because several hyperlinks are obsolete and no longer exists, it would make errors and warnings during the rebuild process, so the update was needed.

4.3.1 Dunfell Version /25/

The latest release of the Yocto Project is version 3.1 with codename Dunfell. Dunfell was released in April 2020, and currently it has the longest long-term support provided by the Yocto Project which lasts until May 2024. The Linux kernel used for Dunfell is Yocto's Linux-yocto kernel version 5.4 which is widely used at the moment.

4.3.2 Preparing the Working Environment and Test with Simple Build

The environment is set up on a Linux Ubuntu 20.04 LTS Oracle VirtualMachine. The machine has at least 50GB+ and install the packages which are:

```
$ sudo dnf install gawk make wget tar bzip2 gzip python3 unzip \
perl patch diffutils diffstat git cpp gcc gcc-c++ glibc-devel \
texinfo chrpath ccache perl-Data-Dumper perl-Text-ParseWords \
perl-Thread-Queue perl-bignum socat python3-pexpect findutils \
which file cpio python python3-pip xz SDL-devel xterm
```

After that, the /poky folder branch dunfell was cloned from the Yocto source code. Following instruction to start building the image, a simple core-image-minimal was created first to test the build environment /26/. Before bitbake core-image-minimal, /conf/local.conf and /conf/bblayer.conf information was changed so the result image will be hddimg and the machine intel-corei7-64 was used by adding meta-intel layer.

If the building directory was named when running “source oe-init-build-env” as “build” then the resulted hddimg will be stored in “build/tmp/deploy/image/intel-corei7-64/core-image-minimal-intel-corei7-64.hddimg”

The simple core-image-minimal was tested by booting it to the KK2040 camera by copying the image to an USB stick: “dd if=core-image-minimal-inte-corei7-64.hddimg of=/dev/sdb” with /dev/sdb is the USB live stick.

4.3.3 Developing the Image

When tested the develop environment was working and the image could be booted to the camera, it was time to add more layers and develop the image further.

To create a custom layer for the company laying inside the camera image, it was named meta-procemex. This layer includes applications and drivers needed to run more application for the camera. At the moment of writing this thesis, a recipe named recipe-visionappster has been created to install the VisionAppster application inside the camera image.

VisionAppster is a platform with features such as app templates, access to AI models with ONNX, easy integration with the HTTP API, and fast and free code readers. VisionAppster has the tools required to make designing, building, and running vision applications simple every step of the way. It saves time, effort, and a lot of trouble. [27]

The VisionAppster team provides a native build for x86_64 architecture and it is an archive va-engine.tgz file. After extracting the archive file directly in root or another folder placed in root, the visionappster engine can be run with the command va-run.

The VisionAppster Engine is a cross-platform runtime for running vision applications. It provides platform-independent apps a consistent, high-performance execution environment. The Engine and all running applivcations can be controlled using an HTTP-based RPC mechanism through a built-in web server. va-run. /28/

Here is the structure of meta-procemex layer:

```
meta-procemex
 |--conf
 |     |--layer.conf
 |--recipe-visionappster
        |--visionappster
                |--visionappster.bb
                |--files
                     |--va-engine.tgz
```

After that, the recipe was built by bitbake command and the image was built when the recipe had been added successfully. The image now contained the va-engine archive file and after booting, the file can be extracted, and the service could be started by: systemctl start va-engine.service && systemctl status va-engine

The progress of developing the meta-procemex is still ongoing, the camera drivers and the Linux kernel have not been configured, yet.

5 CONCLUSIONS

The purpose of this work was to rebuild the working kuvio-distro and get it booted and running with the Procemex camera KK2040 and develop the image with the latest version of the Yocto Project.

Overall, the project has been successful in creating the final image, which is the kuvio-distro that can be booted to the hardware device, and a simple minimal image of the Dunfell version of the Yocto project that can be tested with install applications and services for the hardware.

The building of the Yocto project takes a lot of time and configuration since it requires the knowledge of the Yocto project and its rules, as well as using tools to work with the project. From the base of the camera image now, more future work can be performed to be suitable for the usage purpose.

The Yocto Project and its tools are convenient and functional for creating an Embedded Linux distribution. Everything is assembled from pieces that can be combined as desired so that the resulted system is flexible and allows space for future innovation.

A drawback in this development work was that it took a lot of time to build the process for the first time but now, if there are any problems, there is still the build history to look for any errors. Sometimes the problem can be about an obsolete link and out- of- date source, so it can be laborious to update the files manually. But because it is easy to track down what is going on in the build process, all these problems can be solved right away.

REFERENCES

/1/ Heavy.ai. Embedded Systems. Accessed 05.12.2021. https://www.heavy.ai/technical-glossary/embedded-systems.

/2/ Procemex. Procemex Oy. Accessed 12.01.2022. https://www.procemex.com/.

/3/ Daniel, B. . What Are Embedded Systems? Accessed 22.01.2022. https://www.trentonsystems.com/blog/what-are-embedded-systems.

/4/ Electronic Paper. What are the advantages of Linux embedded operating system. Accessed: 12.01.2022. https://ee-paper.com/what-are-the-advantages-of-linux-embedded-operating-system/.

/5/ Bonesio, J. . 2020. The Major Components of an Embedded System. Accessed 01.02.2022. https://thenewstack.io/the-major-components-of-an-embedded-linux-system/.

/6/ iponwire. Understand Linux boot process. Accessed: 03.02.2022. https://iponwire.com/understand-linux-boot-process-in-2021/.

/7/ T. k. d. community. Linux source code layout. Accessed: 12.01.2022. https://linux-kernel-labs.github.io/refs/pull/187/merge/lectures/intro.html#linux-source-code-layout.

/8/ Prakash, A. . 2020. Linux Directory Structure Explained for Beginners. Accessed: 12.02.2022. https://linuxhandbook.com/linux-directory-structure/.

/9/ Robert, P. J. Getting started with the Yocto Project. Accessed: 15.02.2022 https://www.yoctoproject.org/docs/1.8/yocto-project-qs/yocto-project-qs.html.

/10/ Yocto Project. The Yocto Project manual overview. Accessed: 16.02.2022. https://www.yoctoproject.org/docs/latest/overview-manual/overview-manual.html.

/11/ Tutorial Adda. Introduction to the Yocto Project, Poky, and Bitbake. Accessed: 16.02.2022. https://tutorialadda.com/yocto/what-is-yocto-project-poky-and-bitbake.

/12/ Yocto Project. Reference distribution. Accessed: 17.02.2022. https://www.yoctoproject.org/software-overview/reference-distribution/.

/13/ Yocto Project. Yocto Project quick start. Accessed: 20.02.2022. https://www.yoctoproject.org/docs/2.1/yocto-project-qs/yocto-project-qs.html.

/14/ Otavio Salvador, D. A. . 2014. Embedded Linux Development with Yocto Project.

/15/ Flanagan, E. . The Yocto Project. Accessed: 22.02.2022 https://www.aosabook.org/en/yocto.html.

/16/ Variscite. Hello world recipe. Accessed: 17.01.2022. https://variwiki.com/index.php?title=Yocto_Hello_World.

/17/ The Yocto Project. Yocto Manual. Accessed: 25.02.2022. https://www.yoctoproject.org/docs/2.5/overview-manual/overview-manual.html.

/18/ Linux Foundation® and Yocto Project®. 10 Image. Accessed: 05.01.2022. https://docs.yoctoproject.org/ref-manual/images.html.

/19/ Linux community. Recipes Packages. Accessed: 22.02.2022. http://www.embeddedlinux.org.cn/OEManual/recipes_packages.html.

/20/ The Yocto Project. User Configuration. Accessed: 01.03.2022 https://www.yoctoproject.org/docs/1.8/ref-manual/ref-manual.html#user-configuration.

/21/ Yocto Project. BSP guide. Accessed: 15.02.2022. https://docs.yoctoproject.org/3.1.11/bsp-guide/bsp.html.

/22/ Yocto Project. Yocto Project board support package developer's guide. Accessed: 15.02.2022. https://www.yoctoproject.org/docs/3.1.1/bsp-guide/bsp-guide.html.

/23/ Yocto Project. Yocto Project overview and Concepts manual. Accessed: 16.02.2022. https://www.yoctoproject.org/docs/2.8/overview-manual/overview-manual.html.

/24/ Yocto Project. Yocto Project software development kit developer's guide. Accessed: 14.02.2022. https://www.yoctoproject.org/docs/2.1/sdk-manual/sdk-manual.html.

/25/ A Linux Foundation Collaborative Project. Release 3.1 dunfell. Accessed: 17.01.2022. https://docs.yoctoproject.org/migration-guides/migration-3.1.html.

/26/ The Yocto Project. The Yocto Project dunfell manual. Accessed: 20.01.2022. https://www.yoctoproject.org/docs/3.1/brief-yoctoprojectqs/brief-yoctoprojectqs.html.

/27/ VisionAppster. VisionAppster document. Accessed: 01.03.2022. https://doc.visionappster.com/3.0-beta6/engine/.

/28/ VisionAppster. Version 3.0. Accessed:15.03.2022. https://doc.visionappster.com/3.0-beta6/.

/29/ Rifenbark, S. . 2017. Yocto Project Linux Kernel Development Manual. Accessed: 25.02.2022. https://www.yoctoproject.org/docs/2.4/kernel-dev/kernel-dev.html.

/30/ Swaroop. Embedded systems introduction. Accessed: 27.02.2022. https://www.codrey.com/embedded-systems/embedded-systems-introduction/.