**Advanced Driving Assistance System Using Embedded Linux**

Under the supervision of

**Prof. Neamat Abd-ElKader**

**Chapter 1:** Introduction

* 1. **Abstract:**

Every year, several hundred people are killed due to road accidents, in fact, the National Safety Council estimated around 40,000 deaths and 4.5 million injuries only in the past few years. Unfortunately, it happens often that the driver doesn’t have a long enough time frame to process the given information or take a decision making them responsible for more than 90% of road accidents. Even so if they do have a long time frame to take a decision some external factors might affect their judgment, and these factors include exhaustion from work, or drowsiness from a long trip which often ends up putting truck drivers on drugs just so they can stay focused, not paying proper attention to traffic lights/traffic signs which people always fail to acknowledge that they are only made for their safety, and not to forget, the remarkable decrease in the attention span due to the modern technology and the constant flow of information.

All these reasons where enough to suggest that maybe humans where not to be trusted completely to drive a 3000 pounds piece of metal on their own or that maybe with the modern technology that we have, driving is now beneath the human brain.

This is why car manufacturers are trying to develop technologies that make driving safer, smoother and more comfortable. This is precisely the interest of the ADAS “Advanced Driver Assistance Systems”.

* 1. **Project description:**

In this project, we create our own operating system that is based on Linux using a tool called Yocto (more on that in the following chapters). The reason behind this is to have the best performance possible for our applications to ensure the highest possible accuracy and allow us to run multiple applications without having much issues on the hardware of choice, which in our case is Raspberry Pi 4.

For the applications, we chose ones that the driver will find most beneficial and helpful and these include:

**Object Detection**, which was done using OpenCV which is a huge open-source library for computer vision, machine learning, and image processing and now it plays a major role in real-time operation which is very essential in our project. So by taking live footage from an external camera and passing it to our application, we can provide the use with a lot of valuable information regarding his surroundings be that as it may.

**Lane Detection,** which will also be done using OpenCV by performing some operations on the live footage taken from the camera that involve prospective transformation, threshold-ing ,and edge detection to identify the lanes in the street and detect the current one. The driver will find this feature very beneficial as slight misalignments or deviations in lanes can sometimes happen accidently and might cause some serious accidents, so this application’s job is to detect it and help the driver correct it before it is too late.

**Signs Detection,** which is done by using.

Similar to the previous applications, the live footage will be passed to this one to process and identify any traffic sign in sight and quickly alert the user to follow along.

**Chapter 2:** Introduction to Embedded Linux

**2.1 Types of Embedded Systems**

In this section, we will explore the different types of embedded systems and compare the features of each to the needs of our project.

First, there is the traditional embedded systems or the one called, in layman’s terms, bare metal. A bare metal device is a physical device that is completely dedicated to running a single dedicated application, which could for example could be a thermostat control program. Interestingly enough, this is how computers worked back then before the days of PCs, only one program could be booted at the time and only one single application could run at the time. However, when we compare the capabilities of this system to the need of our application, we will find them lacking as we are expected to run more than one application that require high processing and will need multiple threads and processes.

Next, there is RTOS or Real-Time Operating Systems. An RTOS has a relatively simple design, but, unlike Bare Metal, it can start and stop different processes concurrently and to do so, it does of course require a much stronger hardware to run a scheduler giving it a bit more overhead. The scheduler opens up the possibility of multi-threading that allows us to run some tasks concurrently, but an RTOS is not as powerful as an OS (operating system). The most significant difference is usually memory protection and virtualization. Which makes RTOS a good candidate until we take into prospective the development time and the amount of low level code that will be required.

All which takes us to Embedded Linux, or as some might call it, General purpose embedded system. This type of system requires a much capable microprocessor, with an MMU (memory management unit), and have access to RAM and external memory. Moreover, Embedded Linux has access to multi-threading and multi-processing and many common libraries making the code more portable. And finally, it has access to high level languages and advanced features. Making it ideal for developing a project similar to ours.

In the next section, we will dive through Embedded Linux in a more detailed manner and discuss all of its components and details.

**2.2 What is Embedded Linux?**

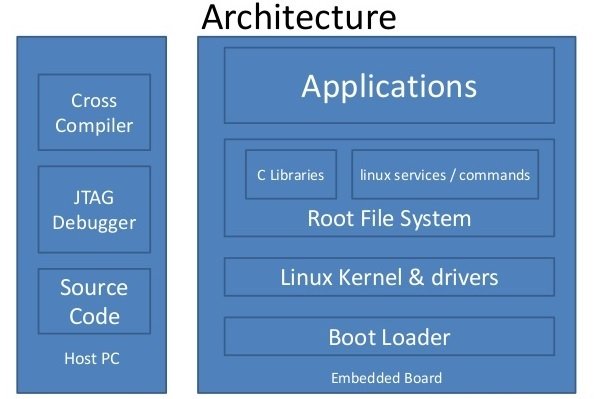
Embedded Linux is a type of Linux operating system/kernel that is designed to be installed and used within embedded devices and appliances. In other words, it’s a compact version of Linux that offers features and services in line with the operating and application requirement of the embedded system. Although it uses the same kernel, Embedded Linux is quite different from the standard operating system. First of all, it gets tailored for embedded systems and, therefore, is much smaller in size, requires less processing power, and has minimal features compared to that of a fully-fledged operating system. The Linux Kernel is modified and optimized as an embedded Linux version. Such a Linux instance can only run applications created specifically for the device.

Embedded Linux also offers its developers with several advantages over other systems such as:

1. Cross-Compilation for any supported platform.
2. Community reflection of common vulnerabilities and exposures (CVE) fixes in updated releases.
3. Deployment to commonly used Linux infrastructure and tools.
4. Modern, cloud-native environment.
5. Broad hardware support.
6. Productive lifecycle through community LTS.

**2.3 Embedded Linux Architecture**

At the most basic level, an Embedded Linux system is one that uses Linux as the operating system that sits between hardware and the application of an embedded device. There are five key components to an Embedded Linux system which we will go through, and these components are:



1. Bootloader.
2. Kernel.
3. Root filesystem.
4. Services.
5. Applications/Programs.
6. **Bootloader**

When the computer is powered on, after performing some initial setup, it will load a bootloader into memory and run that code. The bootloader’s main job is to find the operating system’s binary program, load that binary into memory, and run the operating system, which in our case in the Linux kernel

The bootloader is done at this point, and all of its code and data in RAM are usually overwritten by the operating system. The bootloader won’t run again until the computer is reset or power cycled again.

The bootloader in embedded systems is different from a typical laptop, desktop or server computer. A typical PC usually boots into what we call the BIOS first and then runs GRUB as the bootloader. Embedded Linux systems boot using Das-UBoot or U-Boot for short as the bootloader.

1. **Kernel**

Once the bootloader loads the Linux kernel into memory and runs it, the kernel will begin running its startup code. This code will be responsible for the initialization of the hardware, the system critical data structures, the scheduler, all the hardware drivers, the filesystem drivers, mount the first filesystem, and launch the first program, and more.

The Linux kernel’s main job is to start applications and provide coordination among these applications (or programs, as they are commonly called in Linux). The Linux kernel cannot identify all the programs that are supposed to run, so the Linux kernel starts only one program and lets that program launch all the other needed programs. And this very first program is none other than the init program, or sometimes referred to as just ‘init’. Note that this first program doesn’t need to be in a file called ‘init’, but it often is.

If the kernel for any reason cannot find the init program, the kernel’s purpose is gone and the kernel crashes.

The main takeaway in the Linux kernel for embedded systems is that it is built to run on a different CPU architecture. Otherwise, the way the kernel operates in the same as the typical PC, which is one if its main advantages.

1. **Filesystem**

In Linux, the kernel loads the programs into memory separately, and the kernel expects these programs to be stored on some medium organized into files and directories. This organization is what is known as a filesystem. Linux akin to many operating systems has filesystems on media, which is the data actually stored on a storage medium, and filesystem drivers, which is the code that knows how to interrupt and update the filesystem data on the medium which is often a hardware device like SD cards, or even flash memory.

Unlike Windows, Linux filesystem do not get associated with drive letters, they do get associated with a directory. More so, filesystems can be associated with any directory, even ones that are several layers down in a path, and this association is called ‘mounting’. Linux first starts with an empty directory called / or slash, then during startup, the top most filesystem gets associated with (or in other words, mounted to) this directory, and all the contents of that filesystem appear under / or slash. This topmost filesystem is called the root filesystem.

Linux systems expect the root filesystem to be laid out a certain way. So this filesystem is special and can’t just be some random set of directories and files. This is where directories like bin, sbin and more come from.

Because embedded systems have different hardware constraints, often Linux embedded systems use special filesystem formats rather than the typical EXT3, EXT4, btrfs, or xfs used on desktop or laptop computers.

1. **Services**

When the kernel finds, loads and runs the init program, that program then is responsible for bringing up the rest of the system. At this point, the kernel is no longer actively running and remains to coordinate the sharing of hardware among all the running programs.

There are several init programs available. Regardless of which init program is chosen, this program will launch all of the necessary services and applications that are needed for the system to be useful. This set of services includes setting up networking, mounting additional filesystems, setting up graphical environment, and more.

Under Linux, services are just programs that run in the background. These services were once known as daemon or daemon program, but recently this terminology became less popular.

1. **Application/Programs**

Embedded Linux enables us to run programs in higher level language than that of bare metal embedded. Languages like python, C/C++, and Rust and the most common. The init program is responsible for starting these programs.

Embedded Linux is used to develop care software, and many other examples such as network equipment, machine control, industrial automation, navigation equipment, spacecraft flight software, and medical instruments in general. Even Microsoft Windows has Linux components as part of the Windows Subsystem for Linux or WSL. But perhaps the best example of an Embedded Linux application is Android, developed by Google.

**2.4 Embedded Linux vs. Desktop Linux**

|  |  |
| --- | --- |
| **Embedded Linux** | **Desktop Linux** |
| Linux kernel running in the embedded system product/single board computer/development board. | Linux kernel running on Desktop/Laptop. |
| Real time Linux kernel is used, making the response time real time or deterministic. | Linux kernel running in the desktop or laptop is not real time, the kernel response is not deterministic for response against events. |
| Kernel used in Embedded Linux is the customized version of the original kernel. User configures the kernel as per target processor, components present on the board, need of driver, etc. | Complete version of the kernel is used with all possible drivers and libraries. Whenever any new device or protocol is released then its driver patch is provided by either Linux community or by vendor. |
| Embedded Linux kernel footprint is less, around 1MBs. | Desktop Linus kernel footprint is more, around 100 MBs. |

**2.5 Embedded Linux vs. Bare metal vs. RTOS**

|  |  |  |
| --- | --- | --- |
| **Embedded Linux** | **Bare-metal** | **RTOS** |
| Large overhead compared to other technologies due to scheduler, memory management, background tasks, etc. | Little to no software overhead. | Scheduler overhead. |
| Requires a microprocessor with a memory management unit (MMU) and an external RAM. | Low power requirement. | Requires a more powerful microcontroller. |
| Low direct control of hardware (files or abstraction layers). | High control of hardware. | High control of hardware |
| Multiple threads and processes. | Single-purpose or simple applications, hardware-dependent. | Multi-threading. |
| Multiple complex tasks, like networking, filesystem, graphical interface, etc. | Strict timing. | Multiple tasks, like networking, user interface, etc. |

# Chapter 3: Raspberry pi

## Buy Raspberry Pi 4 Model-B with 4 GB RAM in Egypt- Micro Ohm Electronics3.1 What is Raspberry Pi?

Raspberry Pi is a series of small single-board computers (SBCs) developed in the United Kingdom by the Raspberry Pi Foundation in association with Broadcom.

The Raspberry Pi is a very cheap computer that runs Linux, but it also provides a set of GPIO (general purpose input/output) pins, allowing you to control electronic components for physical computing and explore the Internet of Things (IoT).

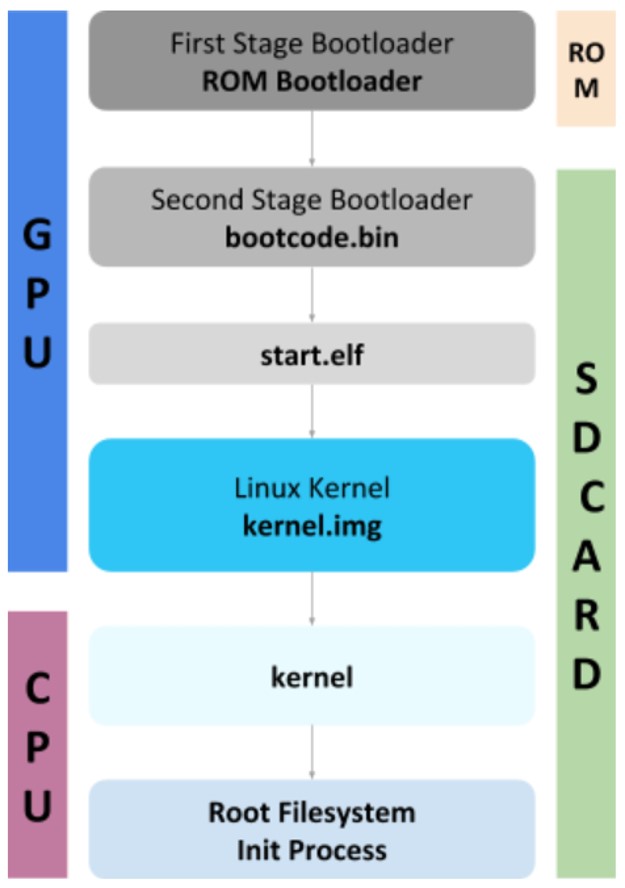
The Raspberry Pi 4 Model B, which is the one we’re using, is the latest version of the low-cost [Raspberry Pi](https://www.techrepublic.com/article/raspberry-pi-the-smart-persons-guide/) computer released in Jun 2019. Raspberry Pi isn't like your typical device, in its cheapest form it doesn't have a case, and is simply a credit-card sized electronic board of the type you might find inside a PC or laptop, but much smaller. It offers ground-breaking increases in processor speed, multimedia performance, memory, and connectivity compared to the prior-generation Raspberry Pi 3 Model B+, while retaining backwards compatibility and similar power consumption. For the end user, Raspberry Pi 4 Model B provides desktop performance comparable to entry-level x86 PC systems.

There are a lot of things we can do with Raspberry Pi, some people buy a Raspberry Pi to learn to code, and people who can already code use the Pi to learn to code electronics for physical projects. The Raspberry Pi can open opportunities for you to create your own home automation projects, which is popular among people in the open source community because it puts you in control, rather than using a proprietary closed system. In our case, Raspberry Pi will be used to run the applications mentioned in the previous chapters, that being object detection, lane detection, sign detection and finally the graphical user interface (GUI) controlling all said applications.

## 3.2 Raspberry Pi 4 specifications

|  |  |
| --- | --- |
| **Processor** | Broadcom BCM2711, quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz |
| **Memory** | 1GB, 2GB or 4GB LPDDR4 (depending on model) |
| **Connectivity** | 2.4 GHz and 5.0 GHz IEEE 802.11b/g/n/ac wireless LAN, Bluetooth 5.0, BLE Gigabit Ethernet 2 × USB 3.0 ports 2 × USB 2.0 ports. |
| **GPIO** | Standard 40-pin GPIO header (fully backwards-compatible with previous boards) |
| **Video and sound** | 2 × micro HDMI ports (up to 4Kp60 supported) 2-lane MIPI DSI display port 2-lane MIPI CSI camera port 4-pole stereo audio and composite video port |
| **Multimedia** | H.265 (4Kp60 decode); H.264 (1080p60 decode, 1080p30 encode); OpenGL ES, 3.0 graphics |
| **SD card support** | Micro SD card slot for loading operating system and data storage |
| **Input power** | 5V DC via USB-C connector (minimum 3A1 ) 5V DC via GPIO header (minimum 3A1 ) Power over Ethernet (PoE)–enabled (requires separate PoE HAT) |
| **Environment** | Operating temperature 0–50ºC |
| **Compliance** | For a full list of local and regional product approvals, please visit |

## 3.3 Raspberry Pi 4 booting sequence

The booting sequence starts with GPU until the kernel is loaded in RAM and then CPU takes overs.

It consists of 5 main stages:

* First Stage Bootloader (ROM Bootloader or RBL).
* Second Stage Bootloader.
* Start.elf.
* Linux Kernel.
* Root File System and init Process.

The booting sequence runs as following:

1. When the system powers on, the ARM CPU is off, the GPU is powered up and starts the booting sequence.

Figure 1

1. The First Stage Bootloader starts executing from the GPU ROM (Boot ROM).
2. Performs some integrity checks on H.W.
3. Searches for the bootable device whether it is SD Card, USB or EMMC (Embedded Memory Card).
4. Loads the Second Stage Bootloader (bootcode.bin).
5. The Second Stage Bootloader starts, and it is responsible for:
   1. Enabling the SDRAM.
   2. Initializing some peripherals.
   3. Loading The Third Stage Bootloader (start.elf).
6. The Third Stage Bootloader (start.elf) loads The Linux Kernel into RAM and reads these files:
   1. config.txt
   2. cmdline.txt
   3. Device Tree Binary File (.dtb)

These files are essential for running the Kernel.

1. By now, the kernel is loaded into RAM and CPU takes over and starts executing The Linux Kernel.
2. The Linux Kernel starts executing until it reaches The Init Process.

## 3.4 Raspberry Pi image structure

The SD Card which is inserted into RPi is divided into two partitions:

* Boot Partition, which is in the format of FAT32, and it contains all the files needed in the booting sequence.
* Rootfs Partition, which is in the format of EXT4, and this is where The Root File System is located.

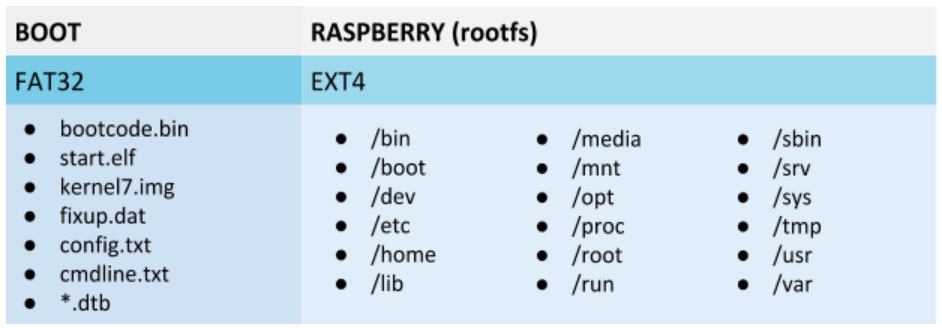


Figure 2

# Chapter 4: Embedded Linux Development

## **4.1 Introduction to Embedded Linux development**

Embedded Linux is a wide field as it can be seen from the previous chapters, hence an Embedded Linux developer has many goals, targets and responsibilities, all which can be divided to:

* Develop and maintain C/C++, python, or rust codes.
* Develop and maintain Linux device drivers
* Develop and maintain Linux environment either through Yocto (most common), or build root, or any other method.

In this Chapter we will discuss the development process of an Embedded Linux Image.

## **4.2 Traditional method for building a customized Linux image**

Rather than the traditional bare metal embedded software development, where we write a C code, build and produce a hex file, then burn the hex file on our target hardware), we create a Linux image that is flashed on the development board through an SD card.

The traditional method for building a Linux image is as it may sound, more of a primitive approach where we have to, as they say, roll up our sleeves and do all the work by ourselves. The development process can be broken down to the following steps:

1. Study the board of choice, and specifically the booting process
2. Build or download a toolchain.
3. Download a bootloader, which is a special piece of software with one sole purpose and that is to load the kernel and hand over the control to it (U-boot is the most popular one).
4. Build the kernel, which is the heart of the operating system.
5. Build the filesystem, depending on our configurations
6. Build our application
7. Plug and play
8. **Studying the booting sequence:**

In the previous chapters, we covered the booting sequence of Raspberry Pi with all of its steps and elements.

1. **Build or download a toolchain:**

First, what is meant by a toolchain, a toolchain is a set of tools that compiles source code into executable that can run on our target device (includes a compiler, kernel headers, binutils, a linker, and run-time libraries).

In order to discuss the importance of the toolchain, we must first understand an important concept, and that is the difference between a cross compiler and a native compiler.

Let’s say that we have a c file called test.c, we compile it using GCC compiler to output test.exe that we run on our host machine. In this case, GCC is called a native compiler because we used it to generate code for the same platform on which it runs.

Say that for the same c file called test.c, we used arm.gcc to compile and output an executable that will run on an ARM target device. In this case, arm.gcc is called cross compiler because it was used to generate an executable code for a platform other than the one on which the compiler is running.

Although we could simply install the compiler on RPI using sudo apt-get but we don’t want do this, we instead want to have/build our own toolchain for better customization. Of course we are not going to build everything from scratch, but instead we will be using tools like crosstool-ng.

As mentioned above, we will be using tools like crosstool-ng to build a toolchain. Let’s see how this process will go.

1. Clone the repository.
2. Search for the suitable configurations.
3. Set the default configurations according to the board we’re using.
4. Apply all the necessary edits through menuconfig

## 4.3 What is Yocto?

**Chapter 5: Installing Yocto and Building the Image**

**5.1 setting up the Yocto environment**

In order to use Yocto Project on our host machine, we must first make sure that the following requirements are met:

1. A host system with a minimum of 50 GB of free disk space running a supported Linux distribution (Ubuntu, Debian, Fedora, etc…)
2. The like git, gcc, python3, and more on the host which are essential for Yocto to generate the image.

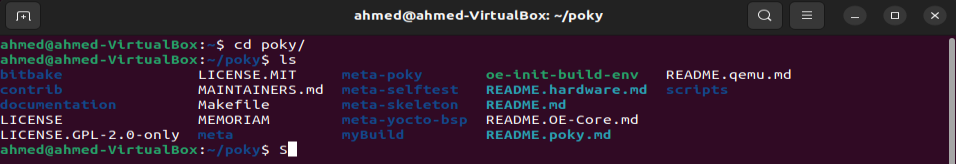
For our project, we will be using Ubuntu 22.04 installed on Virtual Box.

The following command will install said packages based on an Ubuntu distribution:

$ sudo apt install gawk wget git diffstat unzip texinfo gcc build-essential chrpath socat cpio python3 python3-pip python3-pexpect xz-utils debianutils iputils-ping python3-git python3-jinja2 libegl1-mesa libsdl1.2-dev pylint3 xterm python3-subunit mesa-common-dev zstd liblz4-tool

Once the setup is complete, the next step is to install a copy of the Poky repository (which is a reference distribution of the Yocto Project that contains the OpenEmbedded Build System, BitBake, and OpenEmbedded Core, as well as a set of metadata to get you started building our own distro) on our build host using the following command:

$ git clone git://git.yoctoproject.org/poky



The last step in the installation process is to choose the release, which we will be working on, through their corresponding code name, each representing a branch from the Poky repository with its own support lifetime. For our project, we started by using Dunfell, then later, we had to switch to Kirkstone for reasons that will be discussed later.

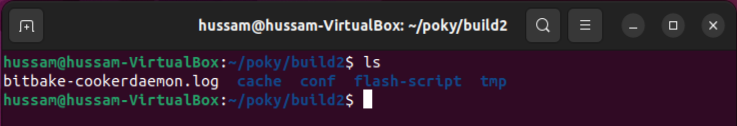
By running the above commands, we now have set up our Yocto Project environment and are ready to create our own distro and layer.

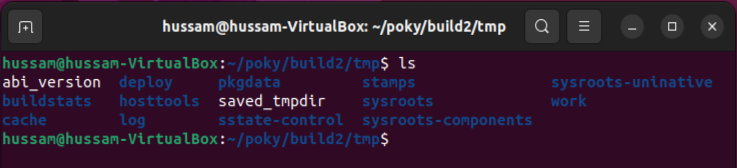
**5.2 Initialize the Build Environment**

From within the poky directory, we run the oe-init-build-env environment setup script to define Yocto Project’s build environment on our build host.

$ source oe-init-build-env myBuild

Running the above command will initialize the build environment by adding some variables related to Yocto to the PATH variable of the currently working terminal and create the build directory, which in our case will be named “myBuild”, and will contain the following folders:



* conf: it contains the used bit-bake layers in bblayers.conf and the build configurations like machine type, package classes, configuration version and many more in local.conf.
* downloads: it contains the downloaded files and packages that will be installed to the custom image.
* sstate-cache: it contains the shared state cache. It will be used later to act as a base for multiple builds. This directory can be moved to another location and specify in the build configuration of the image where it is through the SSTATE\_DIR variable.
* tmp: The OpenEmbedded build system creates and uses this directory for all build system’s output. BitBake creates this directory if it does not exist. As a last resort, to clean up a build and start it from scratch (other than the downloads), we can remove everything in the tmp directory or get rid of the directory completely. Note that if we do so, we should also completely remove the state-cache. tmp also contains the following sub directories:
  + buildstats: stores the build statistics.
* cache: after bitbake parses the metadata (recipes and configuration files), it caches the results tmp/cache to speed up future builds. During subsequent builds, Bitbake checks each recipe (together with, for example, any files included or appended to it) to see if they have been modified. Changes can be detected, for example, through file modification time (mtime) changes and hashing of file contents. If no changes to the file are detected, then the parsed result stored in the cache is reused. If the file has changed, it is reparsed.

* deploy: This directory contains any “end result” output from the OpenEmbedded build process. The DEPLOY\_DIR variable points to this directory.
  + deb: This directory receives any .deb packages produced by the build process. The packages are sorted into feeds for different architecture types.
  + rpm: This directory receives any .rpm packages produced by the build process. The packages are sorted into feeds for different architecture types.
  + ipk: This directory receives .ipk packages produced by the build process.
  + licenses: This directory receives package licensing information. For example, the directory contains sub-directories for bash, busybox, and glibc (among others) that in turn contain appropriate COPYING license files with other licensing information.
  + Images:  This directory is populated with the basic output objects of the build (think of them as the “generated artifacts” of the build process), including things like the boot loader image, kernel, root filesystem, and more. If we want to flash the resulting image from a build onto a device, look here for the necessary components. Note that when deleting files in this directory. We can safely delete old images from this directory (e.g. core-image-\*). However, the kernel (\*zImage\*, \*uImage\*, etc.), bootloader, and other supplementary files might be deployed here prior to building an image. Because these files are not directly produced from the image, if we delete them they will not be automatically re-created when we build the image again.
  + sdk: The OpenEmbedded build system creates this directory to hold toolchain installer scripts which, when executed, install the sysroot that matches your target hardware.

* sstate-control: The OpenEmbedded build system uses this directory for the shared state manifest files. The shared state code uses these files to record the files installed by each sstate task so that the files can be removed when cleaning the recipe or when a newer version is about to be installed. The build system also uses the manifests to detect and produce a warning when files from one task are overwriting those from another.

* sysroots-components: This directory is the location of the sysroot contents that the task do\_prepare\_recipe\_sysroot  links or copies into the recipe-specific sysroot for each recipe listed in DEPENDS. Population of this directory is handled through shared state, while the path is specified by the COMPONENTS\_DIR variable. Apart from a few unusual circumstances, handling of the sysroots-components directory should be automatic, and recipes should not directly reference build/tmp/sysroots-components.

* sysroots: Previous versions of the OpenEmbedded build system used to create a global shared sysroot per machine along with a native sysroot.
* stamps: This directory holds information that BitBake uses for accounting purposes to track what tasks have run and when they have run. The directory is sub-divided by architecture, package name, and version.

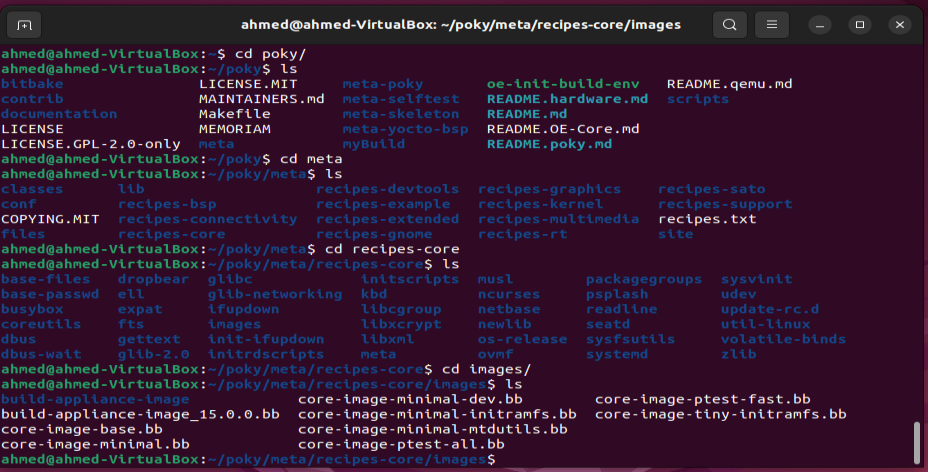
* work: This directory contains architecture-specific work sub-directories for packages built by BitBake. All tasks execute from the appropriate work directory. For example, the source for a particular package is unpacked, patched, configured and compiled all within its own work directory. Within the work directory, organization is based on the package group and version for which the source is being compiled as defined by the WORKDIR.

* work-shared: For efficiency, the OpenEmbedded build system creates and uses this directory to hold recipes that share a work directory with other recipes. In practice, this is only used for gcc and its variants (e.g. gcc-cross, libgcc, gcc-runtime, and so forth).

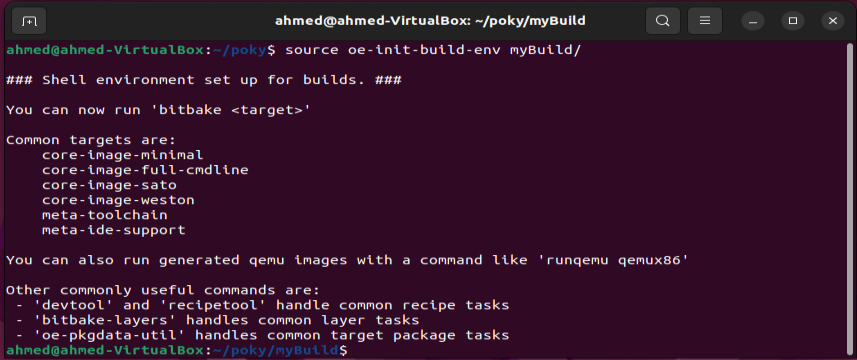
**5.3 Building the minimal image using QEMU**

The OpenEmbedded build system provides several example images to satisfy different needs. When we issue the bitbake command and provide a “top-level” recipe that essentially begins the build for the type of image wanted.

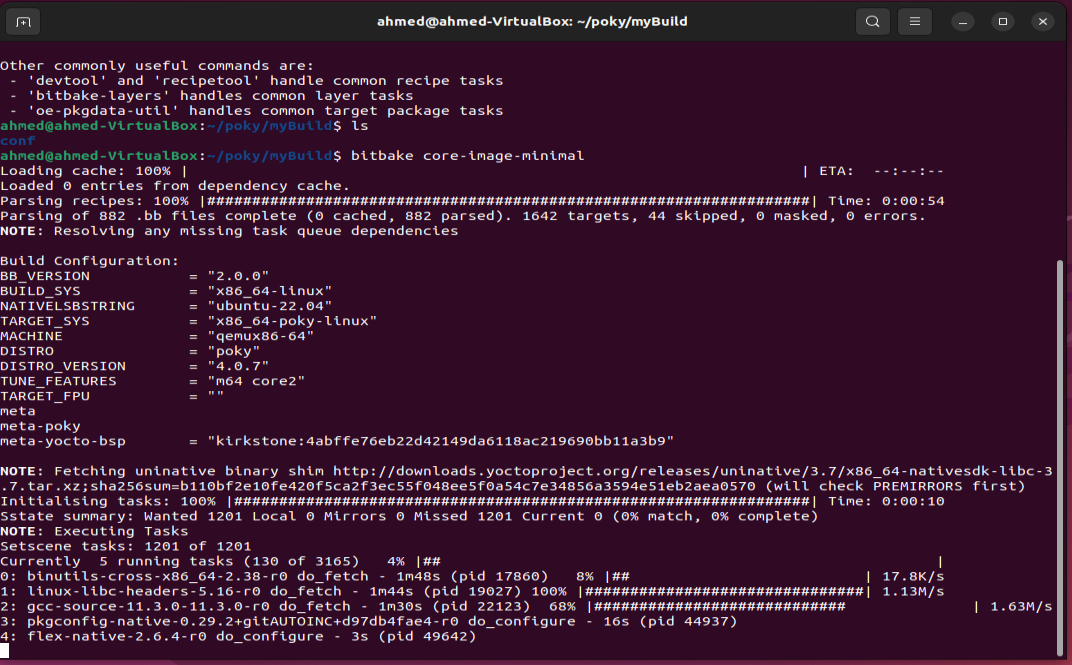
From within the Poky Git repository, we can use the following command to display the list of directories within the source directory that contain image recipe files.



For this example, we will be building core-image-minimal, which is a small image, just enough for allowing a device to boot. To do so, we go to our Poky Git repository, source our build environment through the command

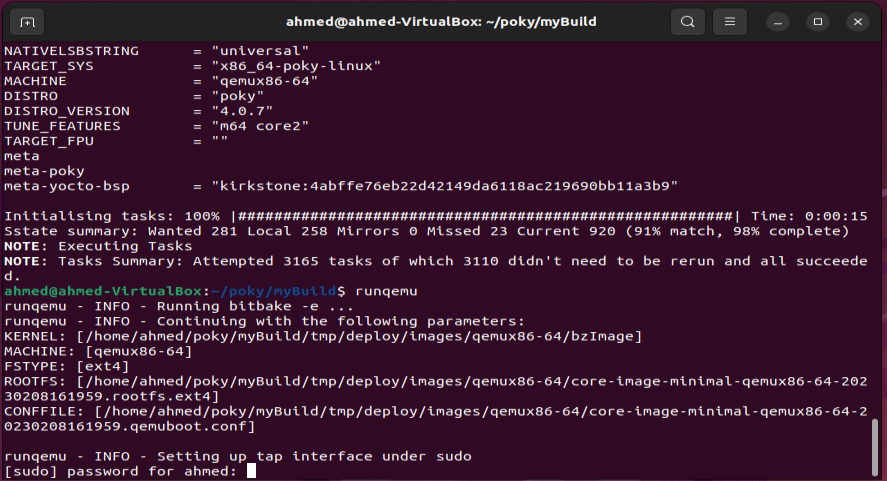


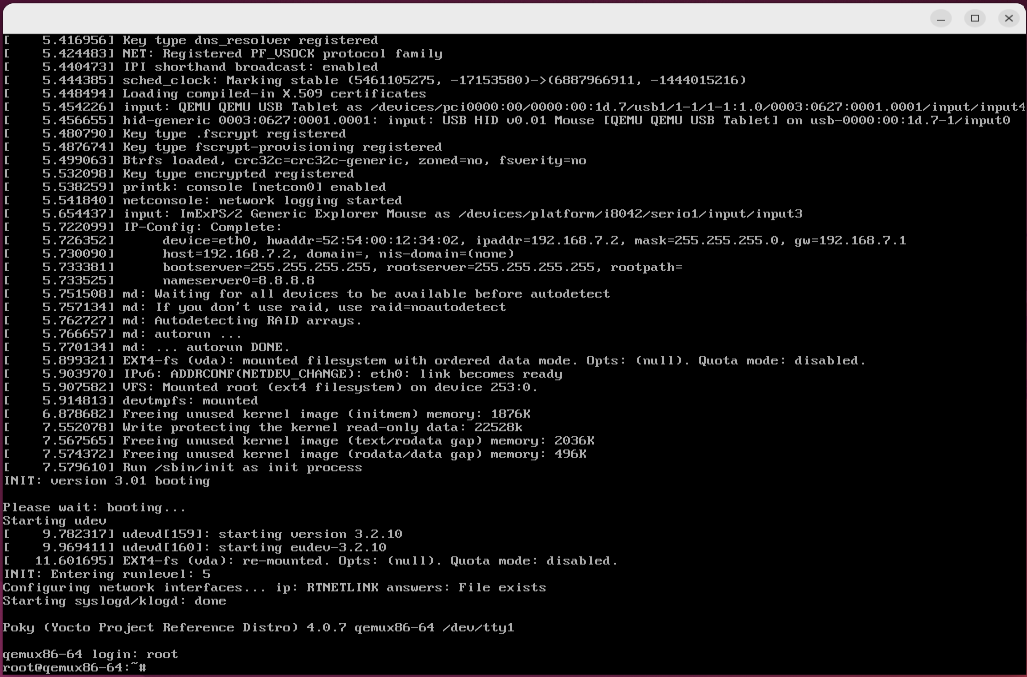
Then we finally run bitbake.



Let’s discuss what QEMU is. QEMU is a generic and open source machine emulator and virtualizer. When used as a machine emulator, QEMU can run OSes and programs made for one machine (e.g. an ARM board) on a different machine (e.g. your own PC).

Now, let’s run our image using QEMU.



  
  
  
**5.4 Creating our own layer.**

Instead of adding the required packages and applications to our image in local.conf file in the build directory which is not a good practice as by using this way any image will be built with these added packages which is not what we want, we will create a layer of our own to be able to add specific packages and applications to it. Whenever an image is built, we can add this layer to the bblayers.conf file which includes all the added layers to the image. This way the added packages and applications are abstracted so that whenever they are needed, we can just include it in the bblayers.conf file.

There are various ways to create a layer:

* The manual way is to use the meta-skeleton folder which acts like an example of a new layer and provides some examples of the recipes to be used.
* The automatic way is to use the bitbake to automatically create a layer:

$ bitbake-layers create-layer meta-mylayer

Text

Description automatically generated

Figure 1

We used the automatic way to create our own layer. This layer contains:

* Multiple recipes and they are organized so that each set of recipes having the same description or configuration will be in one folder as:
  + recipes-core: It contains recipes related to the image.
  + recipes-kernel: It contains recipes related to the kernel.
  + recipes-apps: It contains recipes that are used to build our custom-made applications.
  + recipes-support: It contains some recipes which add some configuration to an already made recipes inside another layer.
* conf folder: It contains the layer configurations and our own custom-made distribution.
  + layer.conf: This file contains the layer configurations like the compatible Yocto releases, the location of the recipes (.bb) files and some other configurations.
  + distro folder: This folder contains the recipes that is responsible for the custom-made distro which will be discussed later.

Text

Description automatically generatedText

Description automatically generated

Figure 3

Figure 2

Text

Description automatically generated

Figure 4

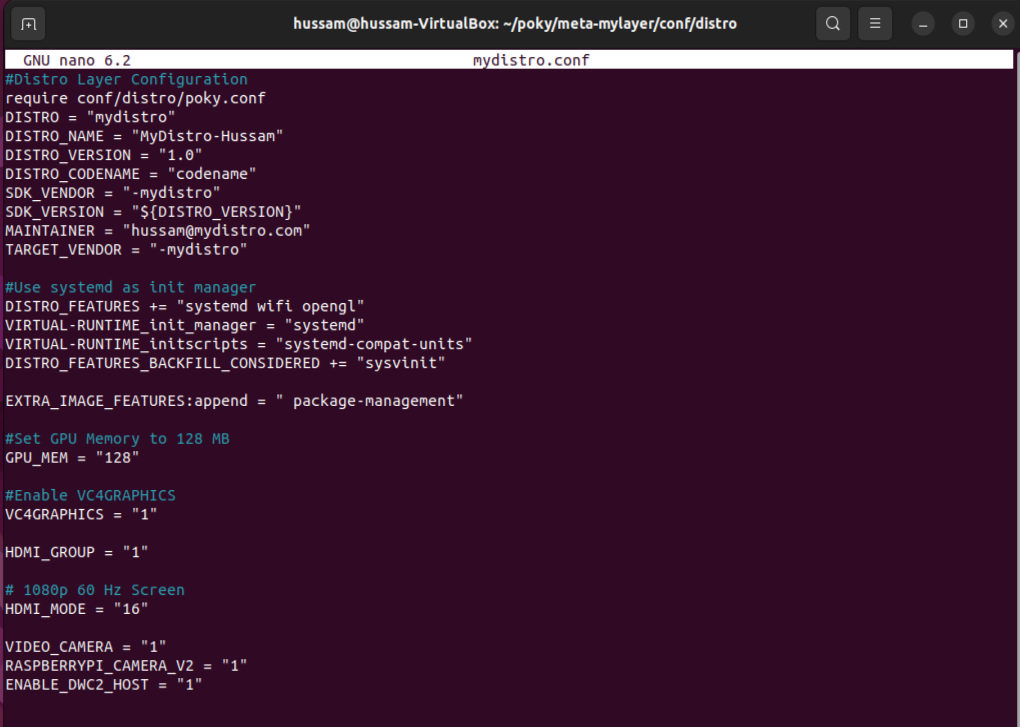
**5.5 Creating our own Linux Distribution**

Now, we will create our own Linux Distribution (often shortened to ‘Linux Distro’) which is basically a version of the open-source Linux operating system that is packed with other components, such as installation programs, management tools, and additional software. Our distribution will be based on Yocto’s Poky distribution.

The custom distro will contain some extra information about the distro itself as (Name, Version, Code Name, SDK Vendor, SDK Version, Maintainer and Target Vendor) each of these information are already present in the Yocto’s Poky Distro so by adding these info in our distro, it overwrites the data already present inside Poky distribution.

In addition to the basic distro information, we will be adding some extra configurations to be included in the distro, like systemd, wifi, opengl, etc.

The distribution configuration file needs to be created in the conf/distro directory of our layer. We will be naming it using our distribution name (e.g. mydistro.conf).

The DISTRO variable in our local.conf, present inside the build directory, file determines the name of our distribution.