Custom Yocto-based minimal Linux image for Raspberry PI, presenting graphical capabilities of IMGUI with OpenGL / GLFW2

By Atanas Georgiev Rusev and Kaloyan Krastev
Triple Helix LLC, https://triplehelix-consulting.com/
Version 1.0, January 25th, 2024

This article and the project have several purposes:

1. Describing how our demo project works, so that you can use it for your needs - from the build to the final usage on the **RPI4B** (Raspberry PI 4B). If you like it, **we will be happy to get a star on GitHub :)**. You can **also use the project for most RPI boards before 4B**! Model 5 is not yet supported by Yocto. The project can also be built for **QEMU**, for which you **don't need HW**.
2. It is also a **primer** for anyone willing to **step into Yocto to make his own build**, for **ANY** of the already supported by Yocto HW platforms.
3. In addition, it presents how this can be done on **Manjaro**, as it is not an officially Yocto-supported **host distribution**. With a few simple installations we can build Yocto projects **flawlessly on Manjaro**. We want to contribute by adding one more distribution to those which are officially approved.
4. As a next point, it shows what **standard SW** we have **integrated in our image**, so that you know for what you can use our image.
5. The project also presents how one can **integrate a GUI application**. In our case it is an IMGUI demo, based on **OpenGL** with **GLFW2** and the **X11** environment.
6. Finally it briefly mentions **Work orders** / **Project development** for **clients**, **Partnerships**, **Change requests**, contributions, and fixes.

Our image also includes a set of **BusyBox** GNU tools, **SSH** connection, **HDMI** setup, and **WiFi** connection script (which saves its values between reboots), which are all described.

Important prerequisite - it is expected, that you have at least some basic Linux knowledge and/or experience (**except for the Introduction**). If you have none or would like to advance, I have written a book, called the **Manjaro Linux User Guide**, a top up-to-date book to learn Linux in 2024!
It covers all general topics from the installation and graphical environments, through available SW, a great Terminal primer, package managers, encryption, Filesystem basics, Storage and backup, Networking and Sharing, Firewalls, Systemctl, Services, Shell scripts and automation, and others.
12 of its sub-sections are shared **for free** as articles! Take a look here:
https://triplehelix-consulting.com/book/.
The main links for the project are:
https://github.com/kaloyanski/meta-thc - main project and layer
https://github.com/TripleHelixConsulting/rpiconf.git - additional layer
https://kaloyanski.github.io/meta-thc/thchowto.html - the developers technical documentation
https://www.raspberrypi.com/products/raspberry-pi-4-model-b/ - Raspberry PI technical documentation

For reference - the abbreviation **thc** stands for **Triple Helix Consulting**.

Table of contents:

1. Non-tech Introduction

What is Yocto, what is your own custom Linux build, when can this be useful?

1. Yocto is an open-source system to automatically build a custom version of the Linux Kernel based on your own custom settings. It also adds a big set of additional tools, libraries, and applications, according to your configuration, and builds them all for custom target HW. For this purpose there are reference projects, so you don't need to know all the stuff, instead take a reference one and investigate and modify only the settings you need.

2. A custom Linux build has the following general advantages:
    a. It uses the best OS kernel, applicable for small devices, PC, server, custom HW, a real time system or anything else.
    b. It profits from the open source ecosystem of Linux, as there are tens of thousands of libraries, modules, frameworks, features and functionalities, so we can choose exactly the ones we need.
    c. This means that if you want a specific graphical environment - you can add it, if you need a custom proprietary application - you can add it.
    d. In addition, each such module can be customized as we wish.

The regular PC distributions are magnificent if you use them for your PC. However, if you want to make a custom robot, embedded device, or anything else - they are not applicable. This is because they are built for x86/64 CPU architecture and come with thousands of modules necessary for a PC to work normally, but typically not necessary on your custom device.

3. When can this be useful?
    a. First - when you **develop a custom device**. Say you want to design a specific infotainment system, new smart watch, or a new smart router. Or even - the **Tesla** car **autopilot**.
    b. You want to custom-tailor the SW on this device to only serve the purpose it was made for and potentially to sell it.
    c. For this you need an **OS**, **libraries**, and lots of other **SW**, on top of which you can add **your specific applications**.
    d. In other words - it is a lot **cheaper** to **use the already existing SW**, and build on top of it.
    e. You also typically want to use SW, on which you have **full control**.
    f. Many times, you make contributions to some parts of the SW and add them to the publicly available SW repositories. Then other people both benefit from your work and may in return find issues and help you fix them a lot faster.
    g. You don't need to reveal the custom additional proprietary application, which you have added.
    h. As many big vendors and companies have noticed these benefits, they often offer free SW layers for Yocto and Linux, so that you can easily use (and buy) their HW and SW.

With all this said - **Yocto is the system that combines all these SW parts, and finally produces a SW image to be flashed on the custom HW**.

2. Yocto in 5 minutes from a technical perspective

Yocto is an **open-source Python-scripts** based set of **tools**. To understand the next points you need to know

and understand what **compiler** and compilation are, and how it all works. To put it briefly, a compiler is the tool which takes all the input source code for a component, the linux kernel, or program of any kind and translates it to executable code – one or a few end-user **applications** and **libraries**.

Here is a short description of each of the Yocto basic tools, concepts, and a few general points in the project:

1. Basic Yocto tool: **BitBake**, which is a task scheduler and executor. **Bitbake** downloads all the sources based on **Metadata** in the form of **Layers** and **Recipes**.

2. **Linux** is only the **kernel**. It is **monolithic**, so it is a single binary. To make a workable OS, we need the additional modules. These are at least a few basic GNU tools, often a shell of our choice, drivers, user software, many times also a graphical environment.

3. All these hundreds of **targets** need different **Recipes (.bb and .bbappend files)**. They are separated into **Layers**, as we need some order and classification, to achieve modularity. Some layers are related to HW, others to the kernel, third ones - to system tools, others to user SW. Each **Layer** has hierarchy. Thus, we can easily change a layer, without changing the other ones. There are also **Classes (.bbclass files)**. Here they are as summarized by Kaloyan:

Yocto File Type	Extension	Purpose
recipe	bb	SW build instructions
recipe	bbappend	SW recipe modification
class	bbclass	Shared instructions
config	conf	Build directives
config	inc	Shared build directives

4. As you probably know, when building an open-source tool, application, or library, it often has **dependencies**. After all, that's the idea - to use the existing SW instead of creating it all by yourself. In this way code is not duplicated and whenever a **fix** in a dependency is done - the other people get it automatically. As all the people work with the same interfaces, this leads to **a standard way of how things are done**. From a project management perspective - **thousands of hours of work are saved** and new projects can be started instantaneously. This **reduces** both the **budget** and the **development time**.

5. **Metadata** - this is the collection of **Recipes**, **Configuration (.conf)** files, and **Class** files, which include the information for **Layers** and **dependencies**, and the git/other repositories links from where to download everything, including the kernel.

6. To work with Yocto means to set up all the **Metadata**, so that once BitBake is started it can correctly build all separate modules, the kernel, and then make a target image for a given target HW platform. The target platform can be a Raspberry PI, ODROID, x86 generic PC, Beagleboard or many others.

7. The **Metadata** also contains a **Toolchain** definition, i.e. the correct compiler and settings for it to compile the sources in the correct format and for the correct CPU architecture.

8. The **Metadata** also contains target **Image** definition. This definition includes basic memory characteristics, filesystem map, paths to put the different libraries and executables, and sometimes instructions to start some programs when the **Image** is in operation.

9. **Poky**: To write all this by yourself is a very long and tough process, which is why Yocto has a reference distribution of the whole toolchain and the basic metadata, called **Poky**.

10. **OpenEmbedded** (OE) is a collection of **BitBake recipes** for thousands of open-source modules, separated into **Layers**, also referred to as a **framework** (practically, a super-library). **Poky** contains at least the **OpenEmbedded**-core layer by default.

11. **BSP layer** - **BSP** stands for **Board Support Package** - the metadata for your HW platform. It includes target CPU architecture, memory, storage definition, peripherals description (like , e.g., Bluetooth, card reader, USB controller on the board, etc.). In our case, we use the **meta-raspberrypi**, as this is our target. However, if we want, we can relatively easily switch to another platform with similar characteristics by changing the BSP layer and modifying some general project configurations.

12. **Targets**, **tasks**, and **steps**. As the tasks that BitBake manages are many (we will investigate them later), we have many intermediate **steps** with intermediate **targets**. The word **target** may be a bit confusing. It is used for: the target HW/platform, the target architecture, the target of a task, a build target, etc. If you are new to these concepts, know that each step has **a target**.

    Following is a generic description of the most important build steps performed by a fully configured Yocto project. It is from the very beginning until outputting the final image to be flashed on the target HW, considering that any build step is building for the target CPU architecture:

    a. Based on the metadata **BitBake** first fetches the sources of all the SW (GNU tools, libraries, applications, the kernel, and all their dependencies).
    b. It then sequentially starts building the dependencies in the correct order, so they are ready for the other SW to be built.
    c. Then it builds the common .so libraries and tools, the higher-level .so libraries and the user applications, and also the kernel.
    d. All the previously built packages are kept in a strict hierarchical structure, so that **BitBake** knows where to find them.
    e. Now **BitBake** starts gathering all the binaries to be put into the image.
    f. It finally prepares the image to be flashed.
    g. As the whole process is long, if an error occurs, when restarted it always tries to continue from where it stopped the last time.

13. **Host machine** is the PC or server, on which the Yocto project is running when compiling a target image.

14. **Cross Compilation** is the process of compiling e.g. ARM binaries on an x86/64 machine. In other words - compiling sources for one type of target machine, on a completely different host machine.

15. **Toolchain** is the set of tools for compilation. This is as a minimum the compiler and linker for the target CPU architecture. In the Linux world it often requires also **make**, and optionally a build system tool like

cmake, meson, ninja, Scons, Autotools and others. By default Yocto doesn't need the additional build system tool, but instead it is a build system itself. Thus Yocto works mainly with **make**, GCC and G++. Still, it supports adding builds for modules based on some of the mentioned additional build systems. The reason is that Yocto doesn't want to limit the build configurations of the modules added to the whole Yocto build.

16. **BitBake** is named like this, as we are "baking" bits to get a final product, just like when we get flour, eggs, sugar and others, and follow a recipe, we can bake a cake.

17. **Documentation** - the Yocto project has an excellent but extensive documentation, which is not easily digestible without guides like this. Still - when you know the basic concepts, orienting yourself in it is easy.

18. Finally - as there are **tens of platforms**, for which Yocto is **already set up**, and for each of them **there is a reference Poky** modification, you can **start immediately** making your build.

3. What do you need on Manjaro to run a Yocto build

Manjaro is not an officially checked distribution for running Yocto builds. However, it only needs a set of tools to be able to do that, and following is the list. Before trying it consider the following:

1. If a given package or tool is already present, and you attempt installing it - **pamac** or **pacman** inform you and just check whether there are updates for the given package (like any other GNU/Linux package manager).

2. If in the future another additional tool is necessary, or when a given required tool is not present - you are usually notified in the Terminal while running BitBake. In this case you simply install the missing one, and re-run BitBake, which will continue from where it stopped.

3. Some of the Yocto-required tools differ a bit in package names between different Linux distributions. This is completely normal due to a number of reasons I will not go into.

The officially supported Distributions are **Ubuntu**, **Fedora**, **CentOS**, **Debian**, **OpenSUSE**, and **AlmaLinux**. However, consider that **only some of their releases** are mentioned: https://docs.yoctoproject.org/ref-manual/system-requirements.html.

The packets we need on Manjaro are:
git tar python gcc make chrpath cpio diffstat patch rpcsvc-proto
Simply call:
$ pamac install git tar python gcc make chrpath cpio diffstat patch rpcsvc-proto

After this you can directly run the bitbake commands.

4. The RPI 4B Platform and the Display

Back in the past I needed a powerful test platform. As Raspberry PI is one of the most famous, with enough basic capabilities, hundreds of example projects, and a great user-base, it was a good choice. To it I added the touchscreen, as I wanted to be able to test GUI touch-based SW for graphical UI. Despite this, it is essential that

Raspberry Pi 4B can be connected to **any HDMI display** via its mini HDMI connector.

The Kuman 7 Inch display offered capacitive touch (which was a requirement for me), and a frame, to which I could attach my RPI. Any similar touchscreen will serve good.

As the Linux graphical environments (including Manjaro for ARM) support touchscreens, this was perfect for my purpose. For the moment our basic Yocto image uses it as a single-point touch device, i.e., a regular mouse.

If you want to add any other custom HW - you need the given HW and to know whether there are Linux drivers/support for it.

5. Yocto and Raspberry PI

The **Yocto RPI project** is **over 11 years old**. It is **mature**, under active development, and with great support! In its manual there is information on how to enable: HDMI, I2C, SPI, UART, USB, CAN, infrared, work with some GPIO, RTC, the Raspberry Pi Camera Module, boot, GPU, Bluetooth, and WiFi!

It supports: **raspberrypi**, **raspberrypi0**, **raspberrypi0**-wifi, **raspberrypi0**-2w-64, **raspberrypi2**, **raspberrypi3**, **raspberrypi3**-64 (64 bit kernel & userspace), **raspberrypi4**, **raspberrypi4**-64 (64 bit kernel & userspace), **raspberrypi**-cm (dummy alias for **raspberrypi**), **raspberrypi**-cm3.

Now you see - it is quite well fit for any test purposes. Check the documentation at: https://meta-raspberrypi.readthedocs.io/en/latest/

6. A few Linux basics

If you have no Linux experience, don't worry. Learning is part of the process (again, in this case you can look at my book - the **Manjaro Linux User Guide**). Here are a few paragraphs if you have no idea.

Linux is only the OS Kernel. This means - by itself it has no tool for anything. It is almost always combined with the GNU tools, a shell (bash, zsh, csh, tcsh or other), other tools and applications SW, and potentially a Graphical Desktop environment to get a Linux distribution.

While with Yocto you **can make** your own distribution, this is **not its purpose**. For such a goal, people use e.g. the Debian distribution creation toolset, or the few others from other Major distributions.

Our image is minimal. It consists of the Linux Kernel, BusyBox, Dropbear SSH, Network drivers (for Ethernet and WiFi), DHCPCD, and a few others. For the GUI part it includes a X11 implementation, a GLFW2 OpenGL implementation, and the IMGUI GLFW2 demo.

Boot

Boot is called the first process started on a computer, before starting an OS Kernel. For a desktop Linux distribution, the GRUB bootloader is used most of the time, and it then starts some Kernel OS (depending on what is available on its storage). As the whole GRUB system is big, complex, and includes BIOS and (U)EFI parts, it is rarely necessary for a microcomputer like Raspberry PI.

Thus, our Raspberry PI configuration uses the default lightweight RPI boot from the **meta-raspberrypi**.

Listing the **/boot/** directory after running the RPI will give you some hints, later I will share some more paths.

Partitions and image

Any storage device (like a PC hard-disk or SD-card), needs data describing: the amount of memory, filesystem type, files table, occupied and available memory,etc. A Partition (covered in my book in Chapter 10) is a separate space on a storage, which has these elements set up.
One storage device may have one or several partitions. For Linux, the main obligatory partition is **root**, and any OS has in addition a small **boot** partition. Often we also may have a separate dedicated **home** and **swap** partitions.
Our image is minimal, and has only the **boot** and **root** partitions.

7. Yocto versions and information, how to know what kernel version do we build from Poky

The main directory, which you will get from our project, is with metadata (there are easy steps for this later). It contains:

```
metadata
├── oe
├── poky
├── rpi
└── thc
```

oe are all recipes, coming from the OE core for application SW and modules.
poky are all recipes, coming from the Poky reference distribution.
rpi are all recipes, related to the Raspberry PI.
thc are all **our custom** recipes.

As Poky is the reference distribution, the **Linux Kernel version is contained in it**. Our build is based on the **Kirkstone** branch, which was the latest **LTS** (Long Term Support) branch, at the moment when the project was started. The branches are described in the following two links:
https://www.yoctoproject.org/development/releases/
https://wiki.yoctoproject.org/wiki/Releases

For some reason, even in their release notes there is no information for the **Kernel versions** integrated in each release. And you will not find this information online. But **there is an easy solution :).**

Either browse online in the respective git repo, or after you download our project, described in the next steps, navigate to **metadata/poky/meta/recipes-kernel/linux/**.
There you will see the **5.10** and **5.15** suffixed **bb recipes** files, so the supported Linux Kernels for the **Kirkstone Poky** release are **5.10** (Super LTS, until Jan 2031) and **5.15** (LTS until Oct 2026).

As Poky always takes LTS releases, this means updates for critical issues and fixes will be provided for a few years. The LTS support dates are available at https://kernel.org/category/releases.html. The corresponding Wikipedia page has a lot more information: https://en.wikipedia.org/wiki/Linux_kernel_version_history.

8. The BitBake build steps

Bitbake is a tool to get the hundreds of recipes, get their relevant sources, build the recipes targets accordingly, and then combine them in an image ready to be flashed on the target HW storage.
It can perform tens of possible tasks, all described at https://docs.yoctoproject.org/ref-manual/tasks.html. On this page we have four sections:
6.1 Normal Recipe Build Tasks
6.2 Manually Called Tasks
6.3 Image-Related Tasks
6.4 Kernel-Related Tasks

Here, I will describe the **most common Normal** ones. Consider that almost all of this section is taken from the official latest Yocto Reference Manual, Version 4.3.999 , on Jan 10, 2024. To make it easier for you, the links are from it, but should for years point to the latest documentation release.

do_fetch is an obligatory command for fetching sources via a **fetcher**. Those can be in a git repository, tarball archive, ftp or http connected server and others, described in a SRC_URI variable. Here is the full list of SRC_URI supported sources and protocols. As there are many possible sources, the possible fetchers are multiple, described here: 4.3 Fetchers.

do_unpack is the command to unpack source code into a target working directory.

do_patch is an optional step that applies patches to sources. Patches are applied to a given module for fixing bugs or security issues, improve performance, change HW or SW compatibility, modify features, etc. They are based on a simple diff file, dedicated to exact source and version. The BitBake recipes will contain SRC_URI variables for such files, and if necessary, they will also be fetched and applied to the already locally unpacked sources.

do_configure is an optional step for configuring sources by modifying build-time and configuration options.

do_compile takes a fetched, unpacked, patched and configured source, and compiles it in the current working directory. As Yocto builds the Kernel, which is mainly written in C and most often compiled with GCC compiler, the GCC/G++/GNU toolchain with makefiles is the default compilation toolchain.

The OE layer has tools for automatic configuration of cross-compilation toolchains, which I will not review in detail, as it is not necessary for our project. You can find more information here: Elinux Toolchains and OE SDK/Toolchain/Application Developer Toolkit. It is also important that external toolchains are supported. Read more about them here, here, and here.

do_install copies compiled files to a dedicated holding area for compiled SW.

do_package, do_package_write_xxx (where **xxx** is **deb**, **ipk**, or **rpm** suffix) are tasks to create a package based on multiple installed binaries, write their additional configuration files, and prepare a .deb or other package.

do_package_qa makes sanity checks (Quality Assurance) on packages files. Check the insane class for more information.

do_depoly: this is usually an additional task run after installation. It copies a given build binary or package to a

user-defined directory.

do_populate_sysroot: takes a set of ready and installed files and copies them to a dedicated filesystem structure. As the target is a Linux system with a given root directory, **sysroot** here refers to this place, which in this case is located in our local build directory.

do_rootfs: an Image-Related task, which creates the **target root filesystem**, including the directory structure, performed as one of the last tasks.

do_build: the default task for all recipes, which depends on all other normal tasks when any of them is configured.

When the build of a program starts, its dependencies are defined, and if not present, first their build starts, and then the final program is also built. A great dependency example is **glibc**, often required for thousands of other binaries to be built.

All binaries (including the kernel) need to be built for the target CPU architecture, which in our case is aarch64 (for an ARM Cortex-A72 CPU).

All these steps result in an iterative process, which first builds the system libraries, then application SW, and the kernel. Most of the fetching is usually done at the beginning, and it takes a significant amount of time depending on your internet connection speed and quality. It also depends on the upstream servers load and capabilities.

After all the fetching is successful, BitBake will start compiling, packaging, sysroot population and deployment to a final directory. When ready, it will call the **do_rootfs** and a few conversion tasks to make a final binary image file.

While working, BitBake provides a current **progress report**, the total number of tasks, and currently executed ones; you will not get lost. It also saves comprehensive **logs**, so if something happens you will always know why and when it happened. The general log for our project is the file:
build/tmp/log/cooker/raspberrypi4-64/console-latest.log.

With all these explanations, when you execute the steps provided in the next section, you will see hundreds of tasks on the Terminal, executed in the span of 1 to 4 hours depending on your internet connection and host machine characteristics. Whenever a fetch is interrupted (e.g., due to internet connectivity issues or a temporary repository server issue), you simply need to restart the general process, and it will continue from where it stopped.

Our project is checked extensively only on **Manjaro**, and many times on **openSUSE**. If the process stops at some of the non-fetch steps, you usually miss some Yocto dependency, and will almost always get a good hint of what is missing. In this case simply install the necessary package, and restart the bitbake process. If you work on Ubuntu, Fedora, Debian, OpenSUSE or other Yocto-supported distro, the dependencies described in the Yocto System Requirements shall be enough.

Once you have made a complete build, any small changes will re-trigger **only the necessary steps**, and not do everything from scratch. This is a common characteristic of most build systems.

The most important BitBake steps for a complete image build are nicely described on this complex SVG graphic by Talel Belhadj Salem: https://docs.yoctoproject.org/_images/bitbake_tasks_map.svg.

9. Fetching the project and building it yourself - super easy, with only a few commands

9.1 Host Machine Prerequisites

You will need **at least** 20 GB of home space to build our project. For our tests we are using relatively powerful machines, in particular at least Intel Core i7-1255U or better, with 16 GB of RAM or more. The build shall work on hosts with less powerful CPU and less RAM, but of course will be slower.
My personal recommendation is to work on a machine with **at least** 4-core-CPU and **8 GB of RAM**, in which case I also recommend having at least 8 GB SWAP space.

Another point: whichever git project you get, no matter the repository, cloning via the **http** protocol is always slower and less error-prone. While the issues are rare, when appearing it can be frustrating. The **git** protocol is a lot faster and more stable.

9.2 Setting up git security key to transfer files from/to github via git protocol (optional)

Setting up this on your Linux host machine is easy, and here is how to do it:

1. Create in your **home** directory a file named **.gitconfig**, filling it with your GitHub-registered email and user name, using on the second and third lines **TAB** for indentation:
   [USER]
   	email=someMail@someServer.com
   	name=YourGitHubUserName

2. You should have in your home directory a hidden directory **.ssh**. If you don't have, create it. Open a Terminal, navigate to your **home** directory, and call **ssh-keygen**. It will ask you for a custom filename and a passphrase for it, which you can all leave empty. Simply press enter 3 times and you will have an automatically generated file **/home/.ssh/id_edXXXXX.pub**, where XXXXX are usually numbers. Open it with a text editor and copy the contents.
3. Now go to a browser, open github.com and login. On the top-right corner click your account circle-icon. From the appeared right-side panel choose **Settings**.
4. On the left side now navigate to the section **Access**, option **SSH and GPG keys**. Click on **New SSH key**, paste the copied contents in the **Key** field, leave the default **Key type** as **Authentication Key**. The title is a custom name for the key, type one of your choice.
5. That's all. Now you will be able to clone projects from github via the faster and better **git** protocol.

To know more about setting keys on **github**, **gitlab**, and **bitbucket** - check this article:
https://dev.to/mkabumattar/git-ssh-keys-for-github-gitlab-and-bitbucket-on-linux-d36

It is essential to point out that any Yocto build consists of multiple subprojects for different modules, libraries, and layers. Thus, even when you make the necessary Git credentials setup, as some of the sub-modules are explicitly set to be fetched via HTTP, they will not be cloned via git protocol. Still, setting up the git access is helpful.
To ease you even more, we have prepared a fetching script, so that you don't need to separately clone repositories and set up the basic directory structure. The fetching script will utilize the git protocol if ssh keys for GitHub are set.

For reference the repositories fetched by our script are:
https://github.com/yoctoproject/poky.git

https://github.com/openembedded/meta-openembedded.git
https://github.com/agherzan/meta-raspberrypi
https://github.com/kaloyanski/meta-thc.git
https://github.com/TripleHelixConsulting/rpiconf.git

9.3 The steps to clone and build the project

1. Make a dedicated directory for the project (I will use the name **yocto-rpi4-thc**) and navigate in it.

2. Download the **metafetch** script from https://github.com/kaloyanski/meta-thc/blob/kirkstone/bin/metafetch.
   $ chmod +x metafetch
   It has a brief help and whenever called will first ask you to confirm the metadata and build directories locations. Ran without arguments it will execute with a default configuration for a **metadata** directory and a **build** directory.

   I always recommend using it with your custom parameters like in the next example. The **-g** argument modifies it to use the git protocol, if SSH keys access is already set up:
   $ ./metafetch -g -b <BUILD_DIR> -m <METADATA_DIR>

   In this way it will create the custom build and meta directories. For me I usually write:
   $ ./metafetch -g -b ~/yocto-rpi4-thc/build -m ~/yocto-rpi4-thc/metadata

3. Now navigate to **/yocto-rpi4-thc/metadata/poky/** and execute:
   source oe-init-build-env <BUILD_DIR>
   In my case this will be:
   source oe-init-build-env ~/yocto-rpi4-thc/build
   This script sets up the current shell build environment by adding several environmental variables, and will also change the current working directory to your **BUILD_DIR**.

4. You can now test your Terminal autocomplete by typing **bitbake** and pressing the **TAB** key. I will provide a short bitbake commands summary in the next sections. Call **$ bitbake-layers show-layers**, to see the currently configured layers in our project.

5. Next simply call
   $ bitbake core-image-thc
   If you use a distribution which is not validated by Yocto as a host system, you will get such warning:
   ```
   WARNING: Host distribution "manjaro" has not been validated with this version of the build system; you may possibly experience unexpected failures. It is recommended that you use a tested distribution.
   ```

   This is only a warning, and based on hundreds of builds we already did, on **Manjaro** in particular there are **no issues**.

   This will be a longer task. It will do all the bitbake steps described earlier, and depending on your system capabilities shall take between 1 and several hours. As already mentioned, if it is interrupted due to fetch issues - simply start it again. If it complains about a missing program (though this shall not be an issue) - install the correct one for your distribution and start the command again. In the worst case, due to fetch

issues, during one of the test builds on my PC it was interrupted 4 times before completing the whole build.

6. Once it is ready, you will have the final **.wic** image for flashing (with the mentioned **/boot** and **/root** partitions) at the path:
   ~/yocto-rpi4-thc/build/tmp/deploy/images/raspberrypi4-64/core-image-thc-raspberrypi4-64.wic.

7. The next step is to copy the image to the microSD card, which we will use on the RaspberryPi4. We usually do this with a simple microSD card USB adapter. On Manjaro (and most Linux distributions), we can use **$ lsblk -fs** to get the currently available storage devices. Call it before plugging the microSD adapter, then plug the adapter, and call it again. You will see the new device, which in my case is **/dev/sdb**.

8. We can now copy the image with the **dd** tool.
   PLEASE BE CAREFUL to which device you copy. Copying to the wrong one can do irreversible damage! As expected, the upper process will destroy any old contents on the microSD.

   Alos - never copy to a partition on the microSD (e.g., sdb1 or sdb2), but always copy to the root device node (in my case /dev/sdb).

   Call **dd** with root privileges like this:
   $ sudo dd if=core-image-thc-raspberrypi4-64.wic of=/dev/<xxx> status=progress

   This command will copy the image, with size of roughly 430 MB.

10. Our project structure and bitbake basics

10.1 The Triple Helix Image

Here is the **tree** listing of our main project `/metadata/thc/meta-thc/`:

```
├── bin
│   ├── burn
│   ├── metafetch
│   └── yoctoinit
├── classes
│   ├── thclass.bbclass
│   └── thconf.bbclass
├── CODE_OF_CONDUCT.md
├── conf
│   ├── distro
│   │   ├── thc.conf
│   │   └── trinux.conf
│   └── layer.conf
├── CONTRIBUTING.md
├── LICENSE
├── README
├── recipes-core
│   ├── dhcpcd
│   │   └── dhcpcd_%.bbappend
│   ├── images
│   │   └── core-image-thc.bb
│   ├── init-ifupdown
│   │   └── init-ifupdown_%.bbappend
│   └── thcp
│       ├── thcp
│       │   ├── .profile
│       │   ├── rpip
│       │   ├── toprc
│       │   └── wifini.sh
│       ├── thcp_0.0.1.bb
│       ├── thcp_0.1.0.bb
│       └── thcp_1.0.0.bb
└── recipes-graphics
    ├── glfw
    │   ├── glfw_3.3.3.bb
    │   └── glfw_3.3.8.bb
    └── imgui
        ├── imgui
        │   └── imgui.ini
        ├── imgui_0.0.1.bb
        ├── imgui_0.1.0.bb
        └── imgui_1.0.0.bb
```

It is essential, that when you used the **metafetch** shell script, it also cloned the additional repository **https://github.com/TripleHelixConsulting/rpiconf** to the build directory, and applied a few simple path changes to the file in **/build/conf/bblayers.conf**. Its purpose was to set up the paths locally on your machine.
In addition, it sourced the script **/metadata/poky/oe-init-build-env**, which resulted in creating a few additional directories in the **build** directory.

10.2 Main files to inspect, and what they are for

Here I will briefly discuss the files in our Layer, and at the end explain a bit the important **rpiconf** project files.

1. Helper scripts

/metadata/meta-thc/bin/
The **metafetch** and **yoctoinit** scripts are here. There is one additional image writing script called **burn**.

2. Distribution definitions

/metadata/meta-thc/conf/distro
Basic Yocto information about the distribution. This is for starters distribution name, version, and a codename. In addition, the file **thc.conf** includes these special directives:

To explicitly **remove** any wayland binaries:
DISTRO_FEATURES:remove = "wayland"

On the next one read more here: https://docs.yoctoproject.org/dev/dev-manual/gobject-introspection.html
DISTRO_FEATURES:remove = "gobject-introspection-data"

This serves to explicitly **include** the **cpufrequtils** package, necessary for correct CPU performance data:
DISTRO_FEATURES:append = " cpufrequtils"

The next one explicitly sets **/home/root** as the root home directory. As it might be set somewhere else, the double question mark operator will apply the value only if it was set nowhere else. Read more about it here: https://docs.yoctoproject.org/bitbake/2.6/bitbake-user-manual/bitbake-user-manual-metadata.html#setting-a-default-value, in sections **3.1.5** and **3.1.6**
ROOT_HOME ??= "/home/root"

3. Dhcpcd

/metadata/recipes-core/dhcpcd/dhcpcd_%.bbappend
This module is an IPv4 and v6 DHCP client daemon. The **.bbappend** recipe commands additional action to the Poky **dhcpcd** module, so that a symbolic link is created for it, etc. However, as we don't include this module explicitly, this additional recipe is not active. It was a working solution in the past. Kaloyan kept it, as an example of how one can append such additional installation instructions.

4. Image Definitions

/metadata/meta-thc/recipes-core/images/core-image-thc.bb
These are the most important general aspects of the image, which include a few important packets and this is where you set your password. Here are a few important points about them.
LICENSE - this part of the Yocto documentation explains licenses, **including** a section on **commercial** ones: https://docs.yoctoproject.org/dev-manual/licenses.html.

IMAGE_OVERHEAD_FACTOR - a variable multiplier to reserve additional free space after the data in the final image. Read more about it here: https://docs.yoctoproject.org/ref-manual/variables.html#term-IMAGE_OVERHEAD_FACTOR.

IMAGE_FEATURES - a variable controlling as minimum the Yocto values described here: https://docs.yoctoproject.org/ref-manual/features.html#image-features. Pay attention to which features we have **removed** and which we have **appended**.

IMAGE_INSTALL - a variable to explicitly install certain packages. Read more about it here: https://docs.yoctoproject.org/ref-manual/variables.html#term-IMAGE_INSTALL. Take a look at what we have added. If you want/need to add a package, depending on whether it is a standard one for the DISTRO_, IMAGE_, or MACHINE_FEATURES, you might need to additionally add it to **IMAGE_INSTALL** for it to be really present on your board. This is the case with **cpufrequtils** - when only added to the DISTRO features, but not here, it is not added to the image and to the manifest. For our image, the manifest is located after a successful build at: **/build/tmp/deploy/images/raspberrypi4-64/core-image-thc-raspberrypi4-64.manifest**.

inherit - a directive to inherit functionality from another class. See what we have added to it, read more here: https://docs.yoctoproject.org/bitbake/2.0/bitbake-user-manual/bitbake-user-manual-metadata.html#inherit-directive

Password - the current default password for the root is **ppp**. We have not set up any other users. You can change it either from the RPI after boot, or if you want to generate another default password - follow the commented instructions in **core-image-thc.bb**. On Manjaro you need to install the **whois** package, which contains **mkpasswd**. Then call **$ mkpasswd -m sha256crypt <somePassword>**. You will get a string like this:

```
$5$cBuwR5SHr226t.rA$opdETNF8n9edBIBcFnbAWv2SDkiZjaIV.97bETcI.6A
```

You have to edit the corresponding **PASSWD =** , adding backslashes before the dollar signs in the beginning.

REQUIRED_DISTRO_FEATURES is a command to enforce the check of certain packages, as explained in the Yocto Glossary.

5. init-ifupdown

/metadata/meta-thc/recipes-core/init-ifupdown/

This recipe served in the past to add the instructions **auto wlan0** for enabling the WiFi upon boot. It is now done in a better way. We kept the recipe for us.

6. thcp

/metadata/meta-thc/recipes-core/thcp/

This directory contains at first level three versions of our install recipe. It presents how Bitbake will take only the one with the highest revision. It also presents the standard way recipe revisions are written after its name. Consider that by convention a recipe shall always be written in small letters. If you write your own recipes in the future read these links:
https://docs.yoctoproject.org/dev-manual/new-recipe.html#storing-and-naming-the-recipe
https://docs.yoctoproject.org/contributor-guide/recipe-style-guide.html#version-policy

https://docs.yoctoproject.org/dev-manual/new-recipe.html#properly-versioning-pre-release-recipes

The **thcp** subdirectory has several files to be installed on the target. They are self explanatory. Still - **toprc** is the **top** process explorer configuration. **rpip** is for the ip configuration.
.profile sets the init functions upon boot. The **wifini.sh** is the script for the WiFi configuration.

7. glfw

/meta-thc/recipes-graphics/glfw
This directory contains the **glfw** OpenGL library configuration.

8. imgui

/meta-thc/recipes-graphics/imgui
This directory contains the **imgui** example application demo configuration.

9. rpiconf/bblayers.conf

Here we reach a file, which shall obligatory be present in the **build/conf/** directory of a project in . In it there is a listing of the layers to be included in a build. It is fetched by **metafetch**, and then as mentioned edited to match your personal paths configuration, according to the parameters you have provided to **metafetch** when running it.
A sample **bblayers.conf** file is available in **metadata/poky/meta-poky/conf/**.

10. rpiconf/local.conf

This is a second file, which shall obligatory be present in the **build/conf/** directory of a project in . In it the most important parts for you are the build configuration directives. You shall not change the **CONF_VERSION** - this value comes from **Poky**. The **DISTRO** is just a name.
The **PACKAGE_CLASSES** defines which of the 3 possible packages to be used internally when building the project. We use **rpm**.

The **MACHINE** directive is essential. It shall be defined only in **/build/conf/local.conf**. The Yocto Glossary link describes the list of default **MACHINE**s, which come with it. If you change it - you can build for the **QEMU** machine emulator and test our image locally on your computer. Check the internet for startup guides and its documentation to learn how to use it.

Most of the rest of the settings are standard, we may remove some of them in the future if their default values serve us good enough. Check the Glossary for more information.

The private host build configuration is as Kaloyan needs specific setup on different machines, these are custom settings you should not care about.

It is essential, that as our project is tested both on RPI and on QEMU, Kaloyan has made an explicit require directive for an **.inc** file.
A sample **local.conf** file is available in **metadata/poky/meta-poky/conf/**.

11. rpiconf/*.inc

The **.inc** files contain configuration specific for given architecture, and they are created by us, they don't come from the regular Yocto setup. In this guide we care mostly about the **raspberrypi4-64.inc**, which contains settings explicitly related to the **meta-raspberrypi**.

To get information on them you can read the file **/metadata/rpi/meta-raspberrypi/docs/extra-build-config.md**. If you want you can also call **make** in the directory containing this file and check the other files in the directory. In any case this **.md** is great, as it explains how to enable different modules and change their settings. For me the highlights in it are: HDMI, Camera (if the module is connected), kgdb, U-boot, SPI, I2C, UART, USB host, Radio Module, CAN, infrared, GPIO-shutdown, and RTC.
Some of these settings directly influence configurations later set in the resulting RPI image in **/boot/config.txt**.

10.3 Extracting information for our configuration

Kaloyan has made a helper script to easily extract some essential bitbake data. Here is how to use it, followed by essential project configuration.

First edit our file **/metadata/thc/meta-thc/bin/yoctoinit**. In it uncomment the lines with

```
# export METAPATH=
# export BBPATH=
```

and set them to your metadata and build directories **absolute** paths like this (fixing the path accordingly):

```
# export METAPATH=/home/MyUser/yocto-rpi4-thc/metadata
# export BBPATH=/home/MyUser/yocto-rpi4-thc/build
```

Save the file.
Now go to the Terminal, on which you executed the command **$ source oe-init-build-env** and then called bitbake, and **source** the script like this:
$ source ../metadata/thc/meta-thc/bin/yoctoinit

You can now call the functions from the **yoctoinit** script. It offers five functions – **bbrecipe**, **bbvalue**, **bbpackage**, **bbimage**, and **bbkernel**. Here is what each of them does. Test them all, it is worth it.

Function name	Function description
bbrecipe	Search an existing recipe by name. You can try it with **$ bbrecipe cpufrequtils**. If such is found, you will get the layer name the recipe is in, and the recipe version. Internally it greps the result of **$ bitbake-layers show-recipes**.
bbvalue	**Get directive value for a given layer.** Example after the table. There are hundreds of directives like DISTRO_FEATURES, IMAGE_FEATURES, and MACHINE_FEATURES. Each holds a string describing certain properties. With this one can inspect **ANY** variable mentioned in the Variables Glossary, if it has some values set. It calls **bitbake-getvar**.
bbpackage	Get the list of packages for a layer. Call it like this: **$ bbpackage core-image-thc**.
bbimage	Lists the available image targets for a layer. Call it like this: **$ bbimage core-image-thc**.It currently reports our image **and 21 other standard images!** So you can use our project to build sato, base, rt versions and others!
bbkernel	Calls the kernel configuration interactive tool. You can call it to know how it looks, but except if you know very well what you do, don't change anything.

Here are the **bbvalue** example calls with the following essential Yocto FEATURES variables:

$ bbvalue core-image-thc DISTRO_FEATURES

```
get DISTRO_FEATURES value in core-image-thc
acl alsa argp bluetooth debuginfod ext2 ipv4 ipv6 largefile pcmcia usbgadget usbhost
wifi xattr nfs zeroconf pci 3g nfc x11 vfat seccomp largefile opengl ptest multiarch
wayland vulkan pulseaudio sysvinit gobject-introspection-data ldconfig
```

$ bbvalue core-image-thc IMAGE_FEATURES

```
get IMAGE_FEATURES value in core-image-thc
allow-empty-password allow-root-login debug-tweaks empty-root-password post-install-
logging ssh-server-dropbear x11-base
```

$ bbvalue core-image-thc MACHINE_FEATURES

```
get MACHINE_FEATURES value in core-image-thc
pci apm usbhost keyboard vfat ext2 screen touchscreen alsa bluetooth wifi sdio
vc4graphics qemu-usermode
```

10.4 Bitbake commands summary

bitbake is the main command. Here follows a list of all commands it provides, along with a short description. Each of them offers help. The main you shall use are marked with a green background. The rest suppose advanced usage.
Important note - the **sig** suffix or prefix means in the bitbake context a **signature**, which is practically a **checksum**. They are changed between builds and task re-runs. In this way bitbake knows when to re-run tasks.

Command	Functions
bitbake	With a layer name argument this command builds it. It can build specific recipes as well (i.e., run a specified task), like the common Normal tasks listed earlier. It can as well perform a dry-run, only parse a recipe, and others. Interesting is the **-g** graphviz option, which saves **dependency tree information** in the dot syntax.
bitbake-layers	Works obligatorily with sub-commands. With them it shows layers, recipes, overlaid recipes, appends, and cross-dependencies. It can also create, add, or remove a layer. The -h and --help show for subcommands additional information!
bitbake-diffsigs	Compares siginfo/sigdata files written out by bitbake.
bitbake-dumpsig	Dumps siginfo/sigdata files written out by bitbake.
bitbake-getvar	A Query Variable command, which returns its values. Check the Yocto Variable Glossary.
bitbake-hashclient bitbake-hashserv	Hash equivalence client and Reference server.
bitbake-prserv bitbake-prserv-tool	The PR variable holds the revision of a recipe.This tool and its server are related to a PR Service, so that bitbake handles automatic PR variables increments.

bitbake-selftest	Runs self tests.
bitbake-server bitbake-worker	Both explicitly meant for internal execution by bitbake itself, shall never be used by the end user.
bitbake-whatchanged	print what will be done between the current and last builds. It is usually used with subtasks, as with global builds there are sub dependencies which may be missed. Despite this, what usually is done is: **$ bitbake someRecipe** Do some changes. Then call: **$ bitbake-whatchanged someRecipe** The changes will be printed

11. Starting the board

To do this, you have to have your display connected and put the microSD card in the slot. Then you simply plug-in the power and that's all. Now, if you connect it to an HDMI monitor with higher resolution, you might have a problem with the resolution. The reason for this is that a big HDMI monitor will report its capabilities first, and then the default graphical driver will try to map in the top-left corner a small 800x480 or similar resolution, which may be hard to read.
There are several solutions for this:

1. Connect the RPI HDMI to a monitor with lower highest resolution.
2. Connect the RPI HDMI to a monitor with dual input and Picture-In-Picture (PIP) or PBP modes, if you can set it up correctly, this depends on the monitor.
3. Connect the RPI HDMI to a monitor, for which you can choose the resolution - rare.
4. Connect via SSH, and work with it in Terminal mode. We will see how to do this.
5. If you have a small HDMI display (like our Kuman 7 inch module), you start the RPI with it connected, and then, when it loads, you connect the cable to the additional monitor. In this case it will be displayed on the bigger monitor with better resolution.

If you still have issues - just read the next few paragraphs, after them I will reveal what more can be done for the issue.

Once the board starts, it will first display a log of its initial tasks. It will then attempt to connect with the WiFi, if its SSID and password were already provided during previous boots. During the first boot there is of course no SSID and password saved.
Further it starts the X session, then in it the IMGUI GLFW OpenGL2 demo, and switches the display to it.

Linux (as many people know), always provides a set of virtual consoles to connect to. On a desktop distribution these are usually accessed via **Ctrl+Alt+Fx**, where **x** is a number between **1** and **7** or more. Here is the article for these on the official Manjaro flavors: https://triplehelix-consulting.com/linux-virtual-terminals/.

For our image the default Terminal runs on **Ctrl+Alt+F1**, while the **X11** session with the **IMGUI** demo is accessible via **Ctrl+Alt+F2**. Due to the way it is started it can be stopped only via kill from the default Terminal session on **Ctrl+Alt+F1**. Actually one of our initial purposes was the IMGUI demo to work in a frame-less window which cannot be closed by default.

What you can do is to disable its startup by commenting a line in the file **/home/root/.profile**. You can do this via

the only integrated editor, which is **vi**. In it find the line:
[-n "$DISPLAY"] && /usr/bin/example_glfw_opengl2_cmake
and add a hash sign # to comment it out. Then execute **$ reboot** to restart the board.
To disable it permanently from the image build, open the same profile file in the metadata directory, located at: **/metadata/thc/meta-thc/recipes-core/thcp/thcp/.profile**. In it, find the same line, comment it out, and rebuild the image with **$ bitbake core-image-thc**, and reflash it to the microSD card.

The result will be that the IMGUI demo will not be started, but the X11 session will be started. In this case to start the example in the **Ctrl+Alt+F2** session simply execute **$ /usr/bin/example_glfw_opengl2_cmake**. Now you will be able to close the demo with **Alt+F4** as it will have a frame with a close button.

If you have disabled the IMGUI demo, you can use the **xrandr** X-session control program to list supported resolutions and change them via, e.g. **$ xrandr --fb 800x600**. This resolution is only an example, as the supported resolutions depend on the monitor your system is connected to.

To change the resolution of the main Terminal at **Ctrl+Alt+F1**, you have to edit the script on the RPI with **vi**:
$ vi /boot/config.txt
In it, find the lines related to **hdmi_safe** and **hdmi_mode** and test with different values. We can't recommend specific settings, as we care only about the Terminal at the second Virtual Console, and for the rest of the cases we work remotely via SSH. To know more about many of the possible **/boot/config.txt** settings check the directory **/metadata/rpi/meta-raspberrypi/docs/**, and in particular the file in it **extra-build-config.md**.

Using **halt** and **reboot**

You can always power-off the RPI without any concerns. Still, if there is a hypothetical chance of a SW to write a file on the SD card while doing this may result in corruptions. Thus, it is recommended to use the **halt** command before turning it off.
If you need to restart the board, simply execute the **reboot** command.

Connecting via SSH

For network connectivity, the easiest way is to connect via Ethernet and the local LAN. If you connect to the board with it, call the **$ ip a** and see your Ethernet RPI **eno1 IP address**. Then to connect via **SSH**, simply call from a terminal **$ ssh root@192.168.17.65**, replacing the IP with the one corresponding to your board. As we have only **root** account, type for user **root**, and as password the default "**ppp"** or the one you have set.
For the WiFi setup there is a subsection in the next part. Once it works correctly, again call **$ ip a**, see your **wlp3s0 WiFi IP address**, and connect with ssh in the same way as in the previous paragraph.

After reflashing the image, its SSH signature will change, even if the IP address doesn't. In this case, to connect again via SSH, delete from your local file **~/.ssh/known_hosts** the line for the corresponding RPI IP address, and then call the **ssh** connect command again.

12. Networking and starting the WiFi

Our image contains the DHCP package **dhcpcd** from BusyBox. It adds a command **udhcpc**, described at https://busybox.net/downloads/BusyBox.html. The **wpa-supplicant** package is also available, and it adds **wpa_passphrase** to save the WiFi network password.
The WiFi setup is a bit complex, so Kaloyan has written a script, which automates this and saves the password

between reboots. It is located in our project in **/metadata/thc/meta-thc/recipes-core/thcp/thcp/** and installed during the image preparation. It also uses the complex **iw** utility.

The Raspberry Pi 4B Ethernet works directly by plugging a LAN cable.
To use the WiFi you need to know the SSID (i.e., the name) of the network you will connect to. As the script is integrated in the image (located at **/usr/bin/wifini.sh**), you can call it directly from the Terminal and add the SSID like this:
$ wifini.sh -s <SSID>
For a network named MyNetwork, it will look like this:
$ wifini.sh -s MyNetwork
It will then ask you for a password, and while typing, it will not be hidden with asterisks. Once you type it and press Enter - if the password is correct, the script will save it and reboot.
If called again with the same SSID, it will simply report that it is already connected.

Upon reboot you will be able to connect with the Dropbear **ssh** via the WiFi.

To get your Network settings call **$ ip a**.

13. The SW provided by our image

Our image contains multiple modules. I will try to briefly describe the most important ones.

The full list of binaries integrated in the image with their versions is located in the same directory as the image itself. It is in a manifest file with the same name as the image, but with extension **.manifest**:
~/yocto-rpi4-thc/build/tmp/deploy/images/raspberrypi4-64/core-image-thc-raspberrypi4-64.manifest

This manifest lists all binaries, libraries, and Kernel modules, and for the **kirkstone** branch that we build has 2018 items in the list. Without all the kernel modules, the items are ~278. They include **busybox**, a set of **alsa** modules, **dbus**, **glibc**, **e2fsprogrs**, **file**, **init** modules, **libgl-mesa**, **libudev1**, **matchbox-terminal**, multiple **X11** modules, multiple **xorg** modules, and many others. Some may not be necessary, but for the moment we don't plan some cleanup, as we don't need such. Following are a few words on some of them, starting with **busybox**.

1. **BusyBox**

   It combines tiny versions of many common unix utilities into a single executable. Check its basic description at https://busybox.net/about.html.
   It currently has (among others) some of the most essential GNU tools:
   ls, mkdir, cp, cat, mount, sh (the Bourne Shell), ash (the Almquist Shell) grep, mv, rm, rmdir, ps, pwd, ip, umount, uname, time, top, touch, ln, unlink, wget, which, stat, time, timeout, who, whoami, free, and the basic pagers more and less.

   To get all the commands, which are integrated into the integrated **busybox** version, simply type its name on the Terminal:
   $ busybox

   Most of its commands provide basic help with **-h** or **--help.**
   You can also check the page https://busybox.net/downloads/BusyBox.html.

In addition, listing the contents of the **/bin/** directory is equally great. It will show you links of most of the programs to busybox, and a few others like **e2fsprogs**.
Debian, Arch, Manjaro, and probably many other distributions, have linked **/bin/** and **/sbin/** to **/user/bin/**. As this is typical mostly for desktop distributions, there are no such links in our image. The reason is that in order to be minimal, it relies on the classical structure. Thus, you can also list the contents of **/sbin/** and **/usr/bin/** to see more programs and commands.

For the curious: As revealed in Chapter 9.3 of the Manjaro Linux User Guide, for decades Linux kept binaries in different locations. Here is a quote for the **/sbin/** and **/usr/** directories:

"/sbin: Originally, this was for system binaries and shall contain the init directory, and binaries only run by the root user. In the case of Manjaro, it is a link to /usr/bin.
…
/usr: This is usually the location of all directories for executables, libraries, binaries, and installation files related to and installed by the current user. In the distant past, the /usr/bin directory was on a separate HDD, which was expensive, small, and contained all critical binaries for running the system. When placed there, the most frequently executed binaries were fast to reach and execute. Back then, many other infrequently used directories were on a slower disk. Nowadays, disks are big and fast, and such separation is unnecessary. As a result, Debian and Arch Linux merged the usr directories with the system directories, and Debian called the operation UsrMerge. Currently, many Arch and Debian derivatives use the same approach. This is why bin links to /usr/bin, lib links to /usr/lib, lib64 links to /usr/lib, and sbin links to /usr/bin."

2. Shell

The default shell on our image is the BusyBox implementation of the Bourne Shell, or **sh**. BusyBox also provides the Almquist shell **ash**. Keep in mind that these are limited versions of the originals, so they might lack some features.

3. Init system

Our image uses sysVinit, based on the regular **poky.conf**. There are several more options, all available in **metadata/poky/meta-poky/conf/distro/**. We haven't extensively tested the other options, as they require different system settings. If you want to use them, we kindly ask you to fork our project, so that all future users can benefit from such developments.

Their specifics are as follows:

Config file	Init Manager	Standard C Library
poky.conf	sysVinit	glibc
poky-bleeding.conf	sysVinit	glibc
poky-altcfg.conf	systemd	glibc
poky-tiny.conf	mdev-busybox	musl

For reference the related recipes are **meta-raspberrypi/recipes-bsp/bootfiles/rpi-bootfiles.bb** and **meta-raspberrypi/recipes-bsp/common/raspberrypi-firmware.inc**.

4. Libraries

To see all the .so libraries like libc, libm, libmount, libnss, libtinfo, libudev, libusb, libz, modprobe, etc., list the directories **/lib** and **/usr/lib**.

5. X11

X11 is the Graphical Environment System, described well on its Wikipedia page. Without such a system, no GUI SW can run. It's easily spotted in the **top** listing, by looking for x11 and xorg in process names and paths.
It is developed by the Xorg Foundation. As it is old (in development since 1984), a new one, called Wayland, is currently in development (also by the Xorg Foundation).

6. Avahi

The Avahi Daemon is used for Zeroconf Network configuration. You can see it in the **top** listing.

7. Bluetooth

Our configuration includes a bluetooth daemon which is running, but we haven't used or configured it. It is based on the **meta-raspberrypi pi-bluetooth** module:
metadata/rpi/meta-raspberrypi/recipes-connectivity/pi-bluetooth

8. Dropbear SSH

Dropbear is a minimalistic (for Embedded Systems) SSH-compatible server and client, designed to replace full OpenSSH implementations. Thanks to it you can connect with ssh.

9. Matchbox-terminal

This is a part of the Matchbox packet, its binaries are located in **/usr/bin/**.

10. wpa_supplicant

This module is used for the WiFi authentication.

11. Looking for other modules, present or not.

We have seen the Manifest file, binaries directories, and **top** as ways to inspect what is present on our system. The Layers we have integrated offer more modules, not to mention the ability to add yours. We have excluded some of them so as not to overload our image. Others come either by our choice or as dependencies from other modules.

Apart from these, another way to look for other modules is to grep recursively in the whole metadata directory after build and to inspect the build directory. This is how without much hustle we found that the **cpufrequtils** module is already present in the **oe metadata**. I will explain what this module is for in the Processor speed subsection. Apart from **grep**, you can always use the recursive search of **Kate** or any other rich GUI editor.

12. Logs

As any standard Linux system, our image contains log daemons. The default running log daemons are **klogd** and **syslogd**. You can list the **/var/log/** directory and inspect the **Xorg.0.log**, **boot**, **dmesg** and **messages** logs. We usually do this with **$ cat LogFilePath | less**.

13. Kernel

As mentioned earlier, our image is built with Linux Kernel 5.15. With **$ kmod list** you can list the currently loaded kernel modules.

14. Editor

The image comes with **vi** as the only editor. If you don't know it, it might be a bit hard at first sight. On the other hand, once mastered it is quite powerful. Find a simple **vi** tutorial and cheatsheets, they always help. Here is an example: https://www.atmos.albany.edu/daes/atmclasses/atm350/vi_cheat_sheet.pdf.

15. Clock

The RPI comes without a battery-powered RTC clock. Instead, one needs to add one if necessary, or synchronize via **NTP** and the **hwclock** command. As for the moment we don't need them, we haven't done or tested their setup.
BusyBox provides the standard commands **hwclock** to query and set the system Real time Clock (RTC), **rtcwake** to enter a system sleep state until specified wakeup time, and **adjtimex** to read and optionally set system timebase parameters. At the moment the **hwclock** command returns the error:
`hwclock: can't open '/dev/misc/rtc': No such file or directory`

Search the metadata directory for **hwclock** and inspect potential modules to be added to the **thc** layer.

16. Processor, CPU Speed, and HW specification

The RPI 4B comes with a Broadcom BCM2711 Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.8GHz. The HW specs of the board are available at: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/.
There is one issue with the RPI layer, resulting in getting the wrong CPU model when calling **$ cat /proc/cpuinfo**. Its output reports CPU **BCM2835** (used in the RPI 1 and Zero modules), and performance specified at the moment of writing as **108.00 BogoMIPS**. This is a specific relative processor performance measurement, but not MHz clock frequency..
When one executes:
$ cat /sys/devices/system/cpu/cpufreq/policy0/cpuinfo_cur_freq
it returns frequency in KHz **600,000**, i.e. **600 MHz**. As working with the numerous items in the **/sys/devices/system/cpu/** directory is not comfortable, we have added the **cpufrequtils** package, which adds the command **cpufreq-info**. It reports current speed 600 MHz, and maximum speed of 1.5 GHz, separately for each core. We haven't tested it yet, but the dynamic Kernel CPU frequency shall work, i.e. when there is high workload the system shall have its cpu clock increased accordingly.

Regarding the mentioned issue with some wrong files and reported CPU model, Lance Simms explains in Part 7 of his article how he overcame the issue:
https://lancesimms.com/RaspberryPi/HackingRaspberryPi4WithYocto_Part7.html.
When we or the RPI team will make a fix cannot be said. But you can take Lance's approach and apply it

yourselves.

17. Adding a module

If you want to add a standard module, already present in poky, cpufrequtils is a great example. It is already in the **meta-oe** layer. I read several articles on the internet about BogoMIPS, understood which source file reports the wrong CPU, and then learned about **cpufreq**. So I looked for it in the metadata files, and learned it exists. With several experiments adding it finally required to put the script only in the **DISTRO_FEATURES** and **IMAGE_INSTALL** directives.
If you want to add any custom application or module: use the examples of how we did it with the **imgui** demo (which has multiple dependencies) or the **glfw** library. For simple modules, look at how **wifi.sh** is added.

14. IMGUI DEMO

Our demo application is a fork of the IMGUI demo library, initially ported by me here: https://github.com/AtanasRusevPros/imgui_aarch64_glfw_openGL2_experiment. The purpose was to extract only one of the possible OpenGL lighter implementations, to build it for Linux as a minimal application, and then to be able to use it as a base for the development of different custom GUIs. IMGUI is a lightweight Immediate Mode GUI framework. The demo application is located in **`/usr/bin/example_glfw_opengl2_cmake`**.
It is now part of the **thc** layer, and is configured in the file **/home/root/.profile** to be started automatically at the end of the OS load. As mentioned, you can disable it via **vi** (or completely from the **thc** layer and rebuild the image).

When started it goes to the second virtual console, accessible by **Ctrl+Alt+F2**. As mentioned earlier, to return to the main Terminal press from the Keyboard **Ctrl+Alt+F1**. The demo is impressive, as it is **only 5.1 MB** (compiled for aarch64) and takes only **30 KB of RAM**. Still it presents **hundreds of functions for:** different sliders, buttons, colors, windows, widgets, tables, columns, synchronization, filtering, input fields, and any regular GUI elements with hundreds of attributes for them. It also displays a framerate, at the moment of writing 30FPS.

You can try all features with a USB connected mouse. In our case we can as well use the Kuman 7 inch Touchscreen, which works in this case as a single-point device.

15. Changing for different HW

We haven't tested this, as we don't have the time and HW resources and need at the moment. However, changing for different supported RPI board with HDMI should require only to change the **MACHINE** value definitions in the project_directory/build/conf/local.conf. Currently it is set to "raspberrypi4-64".
What we have done, however, is to build for **QEMU**, thus this works. In any case you may need to add an additional **.inc** file with any platform-related directives.
Whenever you want to change for different HW - obligatory read the **rpiconf** files clarifications (from the **Main files to inspect…** section).

16. Important Yocto Links

These are all to help you know more on Yocto. Take a quick look, some are related to specific widely distributed microcomputer HW platforms (BeagleBone, Odroid, Khadas VIM, etc).

Yocto Releases list: https://wiki.yoctoproject.org/wiki/Releases
https://www.yoctoproject.org/development/releases/#full

The Yocto Project beginner class presentation:
https://docs.google.com/presentation/d/1hEYedO2vriGYWrUOLKADMXG_3OU1fqivCmKCzbajsQI/edit#slide=id.g104cdce5dff_0_81

Git repository listing, which includes the Raspberry PI meta-layer.
https://git.yoctoproject.org/

Keep in mind that many of the BSP layers' git repositories are not only on GitHub, but also on other servers (like for BeagleBone, Odroid, Orange Pi and many others)! Still on git.yoctoproject.org/, you can find layers for: Qt, java, realtime, intel, arm, aws, xilinx, openssl, dbus, older kernels, related to X11 (Matchbox), and many others.

In addition, most famous HW platforms have their guides on how to use Yocto with their boards. Here are a few:

BeagleBone Black:
https://www.beagleboard.org/projects/yocto-on-beaglebone-black

For Odroid C2 - it is old, but the repository is alive and supported:
https://magazine.odroid.com/article/yocto-on-the-odroid-c2-using-yocto-with-kernel-5-0/
https://github.com/akuster/meta-odroid

Khadas VIM1, 2, 3, and 3L, but not for 4:
https://layers.openembedded.org/layerindex/branch/master/layer/meta-meson/

The OpenEmbedded Layers, close to 200 of them are BSPs:
https://layers.openembedded.org/layerindex/branch/master/layers/#

Nice seminars from Tom King on Yocto, on Youtube:
https://www.youtube.com/results?search_query=tom+king+yocto

BitBake Documentation:
https://docs.yoctoproject.org/bitbake.html
https://docs.yoctoproject.org/bitbake/dev/index.html

Yocto Project Overview and Concepts Manual:
https://docs.yoctoproject.org/overview-manual/index.html

Yocto Project Linux Kernel Development Manual:
https://docs.yoctoproject.org/kernel-dev/index.html

Yocto Project Reference Manual:
https://docs.yoctoproject.org/ref-manual/index.html

BSP Developer’s Guide:
https://docs.yoctoproject.org/bsp-guide/index.html

Yocto Developers Manual:
https://docs.yoctoproject.org/dev-manual/index.html

Toaster Manual:
https://docs.yoctoproject.org/toaster-manual/intro.html

Yocto Project Profiling and Tracing Manual:
https://docs.yoctoproject.org/profile-manual/index.html

Yocto Project Test Environment Manual:
https://docs.yoctoproject.org/test-manual/index.html

Visual Studio Code Plugin for BitBake recipes: Look on the VS Code Marketplace for Yocto BitBake from MicroHobby, they have this site at the moment of writing: https://microhobby.com.br/.

17. License, development, and contribution

Our project is licensed under **GPL2**, just like the Linux Kernel. Thus, you can use it as it is for **any closed source product**, as long as you **don't change the code**. Whatever you use it for, TripleHelix **has no liability**.

If you change it - GPL2 **requires** you to **reveal your changes**. The easiest would be to **fork our GitHub project**. This will be great, as both us and any common users will be able to learn from your advances.

If you want to **develop** anything for fun - we will also be grateful to directly **fork our code**, so that again we and any other potential users are able to learn from your advances.

If you simply delete some parts, and then add to the rest your proprietary layer, **GPL2 requires** you to **reveal the modified part** of our project. We also politely ask to explain why you did the given changes, and how this can be useful to any other potential users, so that we all learn from your experience. And then you don't need to reveal any custom additional proprietary code, layers and applications.

We do this for our love of Open Source, Linux, and Yocto. To know more on why open source is so great, check these short presentations: How Open Source drives revenues for billions and why Linux and open source are so great.

18. Change Requests, Bugs, Contributions, New Features

We accept **bugfix pull requests**, as long as they make sense, but as we have other tasks, these might be a bit slow to process. Still we will do our best to react in a timely manner when possible.

If you have any **contribution** to make (including to this **document**), e.g., for enabling a feature, doing modifications and so on - we would be happy to get them. The idea is to develop the project, and this includes its **documentation**, without which even the best project is useless.

If you have any special or time-consuming **Feature** and/or **Change requests** - if they are not **backed financially** we might not be able to serve them. This simple project took half a year to develop and validate, development

costs need to be considered.

If you have any such requests and would like to discuss **paid improvements** - we are **more than willing to discuss them**. In this case please write in our contact form. You can as well contact me, Atanas Georgiev Rusev, on LinkedIn.

19. Projects and work orders, partnerships

If you are interested to work with us on a project, or have a question for consultancy, please write in the TripleHelix contact form. You can as well contact me, Atanas Georgiev Rusev, on LinkedIn.

20. Copyright notice