

## Custom Embedded Linux



Semester project presented by

**Dylan FREI**

**Computer Science with specialization in  
Embedded Systems**

**May, 2024**

Supervisor

**Florent GLUCK**

Client

**ANTS A.I. Systems**

Caption and source of cover illustration : Picture of the device taken by Dylan Frei.

# TABLE OF CONTENTS

<b>Acknowledgements</b>	<b>v</b>
<b>Abstract</b>	<b>vi</b>
<b>Acronyms List</b>	<b>vii</b>
<b>Illustrations List</b>	<b>viii</b>
<b>Introduction</b>	<b>1</b>
<b>1 Chapter 1 : Bootflow : from power-on to shell</b>	<b>2</b>
1.1 BootROM	3
1.2 PSC-ROM	3
1.3 Microboot1	4
1.4 MB2	4
1.5 UEFI	5
1.6 Extlinux	5
1.7 Kernel	5
<b>2 Chapter 2 : Customizing the System</b>	<b>6</b>
2.1 Prerequisites	6
2.2 Required modification for a custom carrier board	6
a MB1 BTC & Kernel DTB	6
b MB2 configuration	7
c Flashing configuration	7
2.3 Creating a custom rootfs	7
a Testing the filesystem on an SD card	8
2.4 Rebuilding and configuring the Kernel from source	9
a Setting up the cross-compilation toolchain	9
b Getting the sources and compiling	9
2.5 extlinux entries	10
<b>Conclusion</b>	<b>12</b>
<b>References</b>	<b>13</b>



## ACKNOWLEDGEMENTS

*I extend my thanks to Pr. Glück and Mr. Schmidt for their supervision of this project. I also thank Pr. Upegui & Pr. Albuquerque for their understanding and granting me extra time for project completion.*

## ABSTRACT

ANTS A.I. Systems is working on the development of a distributed storage and computing infrastructure based on low-power nodes. Each node consists of a hardware platform including a custom carrier board and an Nvidia Jetson Orin **System on Chip (SoC)**. Nvidia provides an Ubuntu 22.04 image for their carrier board and **SoC** as well as a development kit (devkit). The client is interested in investigating this devkit with the objective of creating their custom embedded Linux distribution tailored for their custom board. First and foremost, the primary aim of the project is to study the structure of the hardware and software components supplied by Nvidia and to understand how they work internally. The second objective is to highlight the changes needed to tailor the embedded operating system to the ANTS customised carrier board.

This project presents the structure of the provided firmware and software present in the **Board Support Package (BSP)**. First, it outlines the bootflow from powering the system to obtaining a user shell. Then, it provides a summary of the necessary changes needed for implementation of a custom carrier board and explores several customizations that might be relevant to the final system.



Candidate :

**DYLAN FREI**

Study programme : ISC

Supervisor :

**FLORENT GLUCK**

**In collaboration with : ANTS**

Semster project subject to a company internship agreement : no

Confidential : no

## ACRONYMS LIST

**.dts** Device Tree Source. 6

**BCT** Boot Configuration Table. 4, 6, 7

**BMPM** Boot Media Processor Manager. 3, 4

**BR** BootROM. 3

**BR-BCT** BootROM Boot Configuration Table. 3

**BSP** Board Support Package. vi, 1, 6, 7, 9

**DTB** Device Tree Blob. 6, 7, 8, 10

**EDK2** EFI Development Kit II. 2, 5

**EEPROM** Electrically Erasable Programmable Read-Only Memory. 7

**GPIO** General Purpose Input/Output. 4

**HEPIA** Haute École du Paysage d'Ingénierie et d'Architecture. 1

**MB1** Microboot1. 4, 6

**MB2** Microboot2. 4, 6

**PMIC** Power Management Integrated Circuit. 4

**PSC** Power State Controller. 3

**PSC-ROM** Platform security controller ROM. 3, 4

**SCRs** Secondary Current Regulators. 4

**SDRAM** Synchronous Dynamic Random Access Memory. 4

**SoC** System on Chip. vi, 1, 3, 4, 6

**UEFI** Unified Extensible Firmware Interface. 4, 5

**VM** Virtual Machine. 8

## ILLUSTRATIONS LIST

1.1	Tegra Bootflow . . . . .	2
2.1	Extlinux Modifications . . . . .	10
2.2	Bootmenu . . . . .	10
2.3	Custom Argument seen in commandline . . . . .	11

### URL references

- URL01 Nivida's Developer Guide | Jetson Bootflow



## INTRODUCTION

Jetson Linux is a specialized version of the Linux operating system tailored for use with Nvidia's Jetson platform, which consists of embedded computing boards designed for AI, machine learning, and robotics applications. The company ANTS A.I. Systems is working on the development of a distributed storage and computing infrastructure based on low-power nodes using these modules.

By choosing this project, I was tasked with investigating the provided [BSP](#) and testing several ways of customizing the system. Understanding and documenting the theory behind the bootflow was also necessary. In order to achieve this objective, Nvidia is providing developers with all the necessary official documentation. Thus, my job mainly consisted in extracting and summarizing this information. The module chosen for this deployment is a Nvidia Jetson Orin Nano and, as such, the infrastructure presented here is specific to this [SoC](#).

This project is part of my final year at [Haute École du Paysage d'Ingénierie et d'Architecture \(HEPIA\)](#). Initially chosen as a semester project, this work faced setbacks due to medical reasons. Additionally, due to delays in the production of the custom carrier board, the project could not progress to a Bachelor project and had to be halted early. As such, this document represents about a third of the usual work that goes into a semester project.

In the first chapter, a theoretical exploration of the various boot components is detailed. Then, in a second chapter, the necessary modifications are explained. Several customization options that I could explore during my time working on this project are also documented.

## CHAPTER 1 : BOOTFLOW : FROM POWER-ON TO SHELL

The Nvidia Tegra Bootflow is made up of multiple components, some of which consist solely of extra security measures. My investigation focuses on the primary components and their role in leading up to the final shell that the user interacts with. An overview of the flow control is provided by Nvidia on illustration 1.1.

The bootflow contains the following main components :

- A BootROM micro bootloader, as in any system. This component loads the first stage bootloader.
- A bootloader called MB1, that corresponds to the first stage bootloader. It initializes the CPU and launches MB2.
- MB2, the second stage bootloader, also Nvidia proprietary, responsible for loading UEFI.
- A UEFI being a modified version of [EFI Development Kit II \(EDK2\)](#). It transfers control to the final bootloader once its functions have been executed.
- The top level bootloader, in our case extlinux, which loads the kernel with arguments specified in a configuration file.
- The Linux Kernel

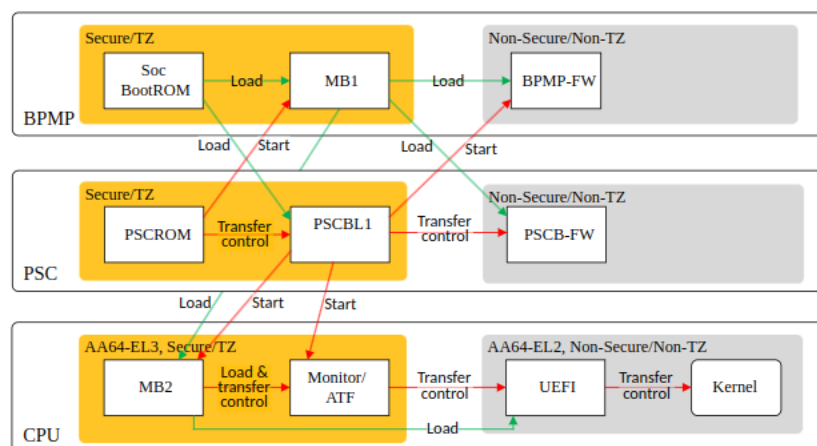


ILLUSTRATION 1.1 – Tegra Bootflow, From <https://docs.nvidia.com/>, ref. URL01

These components and the security components are also divided in 3 "groups" :

- **Boot Media Processor Manager (BMPM)** is the group of processes responsible for the early initialization and configuration of storage media (e.g., SD cards) from which the system can boot. It ensures proper communication with the storage device. BR and MB1 are part of this group, as well as a separate Firmware responsible for coordination.
- **Power State Controller (PSC)** is a component responsible for managing power states and transitions in the system during the boot process. It coordinates the power-related operations of various hardware components to optimize power efficiency. It contains a few components that we shall not explore in details.
- "CPU", a group that isn't exactly an acronym but refers to tasks using the CPU as a mean of booting the system. It mainly refers to the "high level" components of the bootflow, MB2, UEFI, extlinux and the kernel.

## 1.1. BOOTROM

**BootROM (BR)** is a read-only software component hardwired into the SoC. It starts executing as soon as the system is powered on and leaves the reset state. It initializes the boot media and loads a few components from storage - most importantly MB1 and its configuration table -, then halts. BR loads information from a **BootROM Boot Configuration Table (BR-BCT)**, up to 4 copies of which can be stored at the start of the boot media. The **BR-BCT** contains information about the next components including their size, entrypoint, load addresses and hash.

## 1.2. PSC-ROM

**Platform security controller ROM (PSC-ROM)** is a specific hardware component of the SoC that starts running as soon as the processor is reset. It holds the keys that are required for Nvidia authentication and decryption. **BR-BCT** provides authentication and decryption services to BR and controls the next stage of the boot process. This component is mentioned because, while it is BR that loads MB1 into RAM, it is **PSC-ROM** that is responsible for starting it. The presence of this component is essential as it provides Nvidia Owned Keys to the hardware components. As such, modifying the lower-level bootflow is not recommended.

### 1.3. MICROBOOT1

**Microboot1 (MB1)** is the last stage of the **BMPM** part of the bootflow. It initializes part of the **SoC**, including the CPU and performs security configuration. This software is owned and encrypted by Nvidia, depending on the keys from **PSC-ROM**. **MB1** is configurable through its **Boot Configuration Table (BCT)**, which is stored on the storage device. Therefore, **MB1** loads the information about the board from this table, which should be modified to fit the custom carrier board. The functions of the MB1 bootloader are also very clear and well documented. These are :

- Platform configuration, including pinmux, **General Purpose Input/Output (GPIO)**, pad voltage, **Secondary Current Regulators (SCRs)**, and firewalls.
- Initializing **Synchronous Dynamic Random Access Memory (SDRAM)** based on the **BCT**. The Memory **BCT** contains configuration parameters for the memory, specifying settings such as size, timing, and other memory-related details.
- Loading firmware, most importantly a component responsible for initializing the CPU complex.
- Programming the **Power Management Integrated Circuit (PMIC)** for enabling the **VDD\_CPU**. Which means ensuring power to the CPU through the correct pin.
- Creating Memory Caveouts : setting aside specific regions of memory for dedicated purposes, such as graphics memory, system reserves, ...
- Loading MB2.

### 1.4. MB2

**Microboot2 (MB2)** is the generic term Nvidia uses for the second tier bootloader. It comes in two variants : **MB2 Applet** and **MB2** for flashing, development, and Coldboot. Nvidia specifies that the variant of **MB2** depends on the processor it runs on. Based on the information provided by Nvidia, both variants of **MB2** have to do with flashing various components onto the system, either from a x86 host (in which case the Applet runs) or on the system. In the case of a standard boot sequence with everything already flashed onto our SD card, and considering the lack of information, I have to assume that the role of MB2 is merely to load the next stage of the boot process, Nvidia's **Unified Extensible Firmware Interface (UEFI)**.

## 1.5. UEFI

Nvidia's UEFI is a modified version of EDK2, an opensource UEFI software. As such, both sources from Nvidia's version and from the TianoCore community (the original developpers of EDK2) are available. It is likely that this version is modified to fit within Nvidia's own bootflow but information needed to customize it should be available.

## 1.6. EXTLINUX

Our system uses extlinux as a top-tier bootloader. Extlinux is configurable through the extlinux.conf file which contains the boot entries. Extlinux is a Syslinux derivative, which provides lightweight bootloader solutions for various filesystems. However, it is a bootloader specifically designed for booting Linux kernels on systems using the ext2, ext3, ext4, and btrfs filesystems. This could be a limitation as we are pretty much bound to use ext4 for our main filesystem. It was probably chosen for its ease of integration with EDK2. The bootloader can be configured through the extlinux.conf file.

## 1.7. KERNEL

The kernel used for the provided system is a modified 5.10 Linux Kernel. Nvidia provides a toolchain and the sources for its kernel. A real-time version of the Kernel is also available. The regular kernel modules are available and the configuration is saved in a defconfig file. A list of patches to apply is provided in case a previous version of the Kernel need to be used.

## CHAPTER 2 : CUSTOMIZING THE SYSTEM

### 2.1. PREREQUISITES

The easiest way to correctly set up the bootloaders is through the SDK manager. Flashing the SoC once using the SDK manager and the correct storage device ensure proper setup of the firmware before using the command line options. One can then proceed with customization. First, download the BSP and sample rootfs from the Jetson Developer's webpage. The rootfs is derived from Ubuntu 22.04 and must be unzipped in the rootfs directory of the BSP. This includes a modified version of the ubuntu kernel and the corresponding dtb files for building the dtb. The l4t\_flash\_prerequisites.sh script installs all of the dependencies needed for flashing the board.

### 2.2. REQUIRED MODIFICATION FOR A CUSTOM CARRIER BOARD

The goal of this chapter is to understand the necessary steps in modifying the configuration in lower level bootloaders (MB1,MB2) to work with the custom carrier board. It turns out MB1 and MB2 are both stored in an on-module QSPI flash. All the configuration changes necessary to have the module work with a custom carrier board are detailed on the wiki. The wiki specifies that changes need to be made to the following components :

- The kernel DTB
- The MB1 configuration
- The MB2 configurations
- The ODM data
- The flashing configuration

#### a. MB1 BTC & Kernel DTB

To create the BCT for MB1 and the Device Tree Blob (DTB) for the kernel, a single Device Tree Source (.dts) file can be used and compiled. The sample boot configuration tables are available in the BSP under bootloader/generic/BCT. Moreover, the correct dtb file to be passed to the kernel must be specified in the extlinux.conf file. Flashing custom BCT files might require the provided tegraflash.py script. Another script worthy of mention is dtbcheck.py that helps us check the validity of our compiled dtb file before feeding it to the kernel.

## **b. MB2 configuration**

The MB2 configuration should be in accordance with the need of the custom carrier board. The only modification should specify if the carrier board has an [Electrically Erasable Programmable Read-Only Memory \(EEPROM\)](#). The `tegra234-mb2-bct-common.dtsi` file should be modified accordingly :

with EEPROM :

```
cvb_eeprom_read_size = <0x100>
```

without EEPROM :

```
cvb_eeprom_read_size = <0x0>
```

## **c. Flashing configuration**

The flashing configuration mainly refers to selecting our custom [BCT](#) and [DTB](#) files for flashing. It must also include our custom rootfs once available. Flashing is done using the `flash.sh` script. This script is very flexible and can use a large variety of arguments to specify which components should be flashed.

### **2.3. CREATING A CUSTOM ROOTFS**

Technically, any linux rootfs should work on the Nvidia Jetson Orin Nano. This would require having a valid `extlinux` configuration in `/boot`. Moreover, the filesystem binaries need to be generated for the arm architecture. Finally, any Jetpack components desired on the system need to be copied onto the filesystem.

Nvidia, however, provides only the tools to generate an Ubuntu 22.04 and this section explores only this possibility. Older versions of the [BSP](#) provide the tools to generate an Ubuntu 20.04 image. This filesystem contains all of the default tools provided by Canonical. As such, `systemd` is the main daemon and upgrading the system for future releases is possible using `apt`.

The rootfs generation tool works in several steps, one of which uses chroot inside of the filesystem in order to use the host's apt tool to install the packages. This means that rootfs generation is only possible on Ubuntu 22.04 if using the Nvidia provided tools. A [Virtual Machine \(VM\)](#) can be used for this step.

Rootfs generation uses the `nv_build_samplefs.sh` located in `L4T/tools/samplefs`. This script downloads a sample empty filesystem and installs the package list according to a text file. For instance, the file `nvubuntu-jammy-desktop-aarch64-packages` contains the packages for the desktop version of the filesystem. So as to create a custom package list, one of the files can be copied and modified but the naming convention needs to be followed. In general, a name such as `nvubuntu-jammy-[yourcustomname]-aarch64-packages` can be used. After this step, a few more folders, mainly the contents of `/boot` need to be generated using `L4T/apply_binaries.sh`.

```
sudo ./nv_build_samplefs.sh --abi aarch64 --distro ubuntu --
flavor [yourcustomname] --version jammy
```

Furthermore, a few different settings can be set using `l4t_create_default_user.sh` to avoid extra setup time during the first boot. This includes creating a user, enabling autologin, setting the hostname and accepting the license to avoid extra steps when booting for the first time. Any further modification of the filesystem need to be done manually. If one wishes to use a custom [DTB](#) and kernel, or a different extlinux configuration, the correct files also need to be replaced in the `/boot` directory.

### **a. Testing the filesystem on an SD card**

The way I went about flashing and testing the custom filesystem was through the use of an SD card. Nvidia provides a script to turn the filesystem into a flashable blob through `jetson-disk-image-creator.sh`. The image can then be flashed onto an SD card using a gui tool or `dd`. My SD card is mounted as `/dev/sda`.

```
sudo R00TFS_DIR=customfsdir ./jetson-disk-image-creator.sh -
o customfs.img -b ${BOARD} -d SD
sudo /bin/dd if=customfs.img of=/dev/sda bs=1M
```



## 2.4. REBUILDING AND CONFIGURING THE KERNEL FROM SOURCE

### a. Setting up the cross-compilation toolchain

A toolchain for Jetson is provided by Nvidia on the Jetson Developer webpage under "Bootline toolchain". One needs to unarchive the binaries and set the `CROSS_COMPILE` variable to the correct path.

### b. Getting the sources and compiling

The sources can be found on the developer's webpage as "Driver Package (BSP) sources". First, unzip the sources and create an output directory. Then set a few variables. Finally, the default `defconfig` is copied and modification can be made through `menuconfig`. The final step is the compilation of the Kernel.

```
#get and unzip sources
```

```
wget https://developer.nvidia.com/downloads/embedded/l4t/r36_release_v2.0/sources/public
```

```
# From section 2.4.a
```

```
export CROSS_COMPILE=$HOME/l4t-gcc/aarch64--glibc--stable-2022.08-
```

```
l/bin/aarch64-buildroot-linux-gnu-
```

```
#output directory
```

```
TEGRA_KERNEL_OUT=$PWD/kernel_out
```

```
mkdir -p $TEGRA_KERNEL_OUT
```

```
#Kernel configuration
```

```
make ARCH=arm64 O=$TEGRA_KERNEL_OUT tegra_defconfig
```

```
make ARCH=arm64 O=$TEGRA_KERNEL_OUT menuconfig
```

```
#compilation
```

```
make ARCH=arm64 O=$TEGRA_KERNEL_OUT -j10 #number of threads
```

```
#Kernel and dts files are located in :
```

```
/l4t-kernel/arch/arm64/boot/Image
```

```
/l4t-kernel/arch/arm64/boot/dts/*
```

## 2.5. EXTLINUX ENTRIES

All configuration for the main bootloader (extlinux) is made through `extlinux.conf`. This file can easily be modified to allow for multiple boot configurations which can be chosen at boot time. As such, it is possible to have multiple kernels, DTBs or arguments on the system. The easiest way to test this was to create a new configuration based upon an existing one. The arguments are :

- LINUX : points to the kernel image
- FDT : points to the DTB
- INITRD : point to the init ramdisk
- APPEND : boot arguments for the kernel

For instance, we can modify the boot arguments as a proof of concept and add a dummy command line argument :

```
timeout 30
default primary
menu title L4T boot options

LABEL primary
menu LABEL primary kernel
LINUX /boot/image
FDT /boot/dtb/kernel_tegra234-p3768-0000-p3767-0005-nv.dtb
INITRD /boot/initrd
APPEND $(bootargs) root=/dev/mmcblkp1 rw rootwait rootfstype=ext4 mmlib_loglevel=4 console=ttty0,115200 firmware_class.path=/etc/firmware fbcon=map:0 net.ifnames=0 nospectre_bhb videofb=fb:off
console=ttty0

LABEL custom
menu LABEL custom arg
LINUX /boot/image
FDT /boot/dtb/kernel_tegra234-p3768-0000-p3767-0005-nv.dtb
INITRD /boot/initrd
APPEND $(bootargs) root=/dev/mmcblkp1 rw rootwait rootfstype=ext4 mmlib_loglevel=4 console=ttty0,115200 firmware_class.path=/etc/firmware fbcon=map:0 net.ifnames=0 nospectre_bhb videofb=fb:off
console=ttty0 nv_custom_arg=hello
```

ILLUSTRATION 2.1

At boot time, the bootloader asks the user to chose the entry :

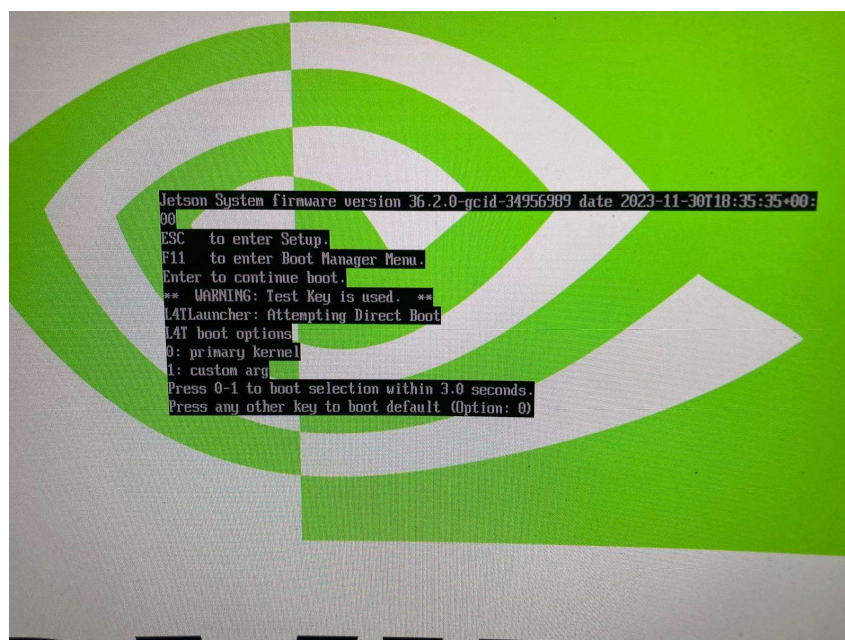


ILLUSTRATION 2.2

We can check that the correct entry was selected using the cat command on /proc/cmdline :

```
dyland@dyland-desktop:~$ cat /proc/cmdline
root=/dev/mmcblk0p1 rw rootwait rootfstype=ext4 mminit_loglevel=4 console=
ttyTCU0,115200 firmware_class.path=/etc/firmware fbcon=map:0 net.ifnames=0
nospectre_bhb video=efifb:off console=tty0 my custom arg=hello bl_prof_da
taptr=2031616@0x271E10000 bl_prof_ro_ptr=65536@0x271E00000
```

### ILLUSTRATION 2.3

The other options such as a different dtb with a dummy device and a kernel with different modules have also been successfully tested.

## CONCLUSION

Nvidia's Jetson software architecture is composed of several key components that we were able to list and explain. While the lower level bootloaders and security components are highly tailored to the Jetson environment, thus hardly replaceable, higher level ones such as the kernel, final bootloader and filesystem are easily customizable using the provided tools.

This work has allowed me to better understand the inner working of a complex, fairly powerful embedded system. It further enhances my knowledge about the theoretical aspects of a cold boot sequence and various aspects of the linux environment. The provided documentation was more than sufficient and sometimes even a bit overwhelming. The research aspect of this project has therefore been tedious at times, with me often losing myself in useless details but has helped me learn to better identify the key points of a wiki. Testing the modifications on an embedded system can also take up a significant amount of time, especially when things don't work out on the first attempt.

Overall, I am satisfied with the results I have reached considering that the project was cut short. Various aspects were explored, but some areas remain a bit obscure, and a few limitations need to be addressed. In particular, because the client is looking for the deployment of a distributed filesystem, the limitations of extlinux as a bootloader need to be investigated. This lightweight bootloader is usually meant to work with ext filesystems.

In conclusion, Jetson Linux presents a compelling alternative to traditional heavyweight servers, especially in scenarios requiring efficient and low-power solutions. The Jetson platform, with its robust processing capabilities and minimal power consumption, is ideally suited for deploying distributed filesystems and various services on low-consumption nodes. This approach not only aligns with modern trends towards greener computing but also enhances the scalability and flexibility of distributed systems. Consequently, Jetson Linux stands out as a powerful enabler for innovative, sustainable, and high-performance distributed computing environments.

## REFERENCES

*<https://docs.nvidia.com/jetson/archives/r35.3.1/DeveloperGuide/index.html> - Nvidia Jetson Linux Developer's Guide - Last viewed on 27.05.2024*