7041CEM Embedded Operating Systems

SID: **14081293**

Module Leader: **Dr Server Kasap**

Total Word Count: **2646** (excluding cover page, table of contents, header-footer, headings, figure labels, one drive link & references)

I hereby declare that the work presented in this document is entirely my own, submitted as part of the requirements for my course award. I confirm that this work has been completed independently, and all sources used have been properly acknowledged.

Table of Contents Page No.

- Introduction……………………………………………………………………………......…….3
- Part 1: Questions……………………………………………………………………….……….4
- Part 2: Embedded System Design…………………….……………..……......……………..………..….12
- One drive link of source code, recipes, makefiles, built image & videos…………………………..………..19
- References……………………………………………………………………………………..…..20

Introduction

The goal of this coursework project is to develop embedded operating systems using the Raspberry Pi, a small single-board computer that is well-known for its educational and prototyping uses. Students work with the Yocto Project to create customized Linux distributions for embedded and Internet of Things (IoT) applications using BitBake and Poky. Students create an application program and loadable kernel module that interacts with physical environments using the HC-SR04 sensor, gaining practical expertise in sensor integration and kernel-level development. Students gain practical experience in system design and application programming for embedded environments, as well as a deeper understanding of embedded systems by working on projects including building kernel modules, customizing Linux images, and assembling Raspberry Pi systems.

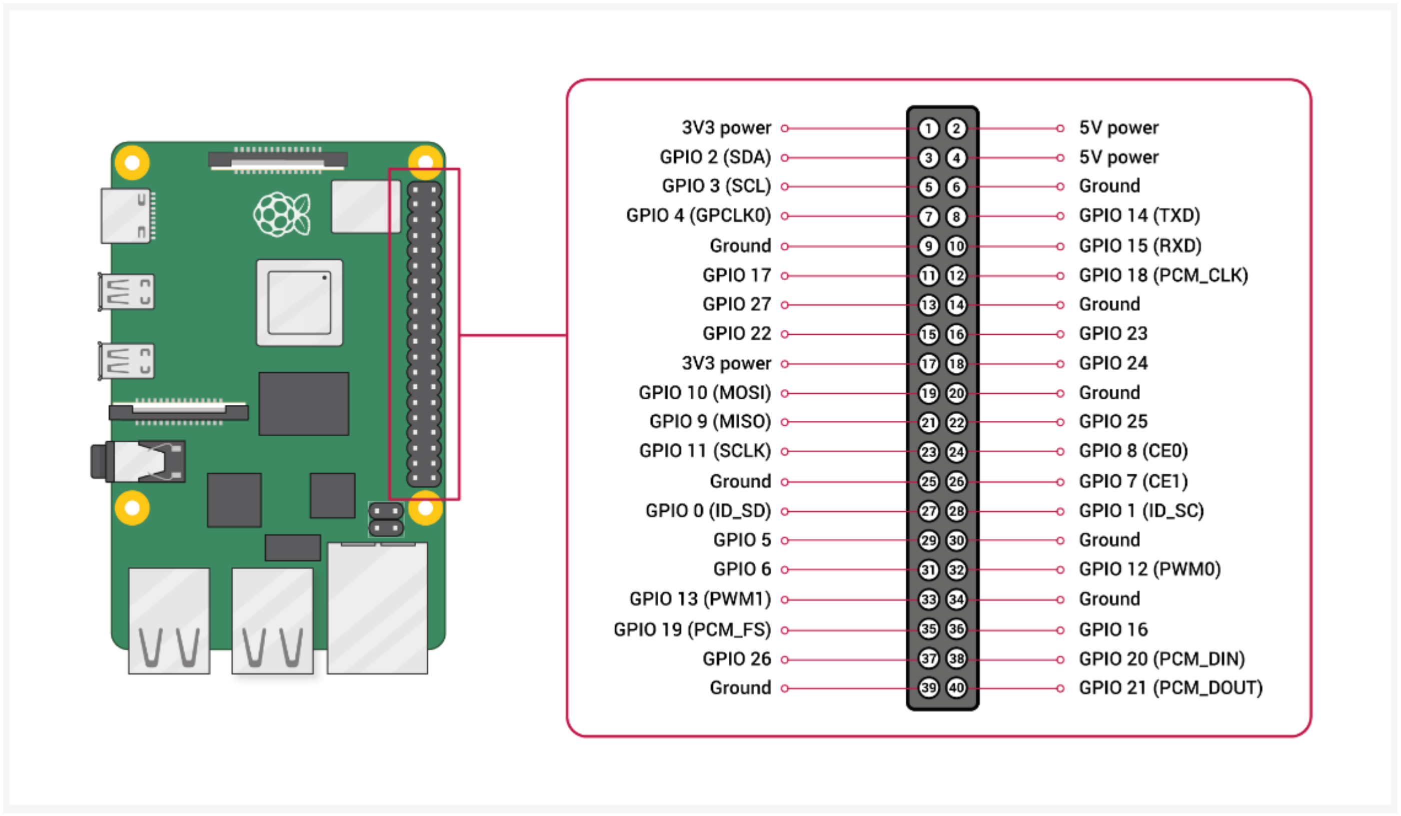
Part 1: Questions // 1489 words

1) Find and summarize the information about Raspberry Pi hardware, its schematics and PCB layouts.

The Raspberry Pi Foundation created the Raspberry Pi board, which is a small single-board computer that is well-known for its affordability and adaptability. Every Raspberry Pi board has an ARM-based processor at its core, which comes in single-core and quad-core varieties. This processor offers enough computational capacity for various activities, from simple computing to multimedia streaming and home automation. Memory options range in RAM capacity from 256MB to 8GB, which allows seamless multitasking and effective data management, and MicroSD cards are commonly used as a means of storage since they are user-friendly and flexible. There are numerous connectivity choices available including USB ports, Ethernet, HDMI output, and GPIO pins. Moreover, many versions come equipped with built-in Wi-Fi and Bluetooth. It's simple to power the Raspberry Pi, all you need is a 5V power source connected to micro-USB or USB-C connection. Operating system support is provided with choices such as Linux distributions and Raspberry Pi OS, which make it simple for users to add new features like sensors, displays, and networking capabilities, which encourages creativity and innovation in a wide range of projects and applications.

Figure 1

Raspberry Pi Schematic



Adapted from raspberrypi.com (n.d.)

Not all functions or parts are included in every Raspberry Pi board version. It depends on variables like cost, power consumption, and intended uses. Every iteration brings improvements, changes, or trade-offs. Thus, the schematics and layout documents for different Raspberry Pi models vary in terms of included components, interface configurations, and circuitry complexity.

2) What type of embedded storage is used for Raspberry Pi? What is the advantage of using it over internal memory storage? Explain the boot process of the Raspberry Pi. *(344 Words)*

Embedded Storage used for Raspberry Pi:
MicroSD cards are commonly used by the Raspberry Pi as integrated storage, offering affordability, flexibility, and scalability. They offer a variety of storage choices for the operating system, applications, and data, with different capacities to meet various requirements. They are still the most popular option for Raspberry Pi users because of their versatility and ease of upgrading, even though some variants come with inbuilt storage such eMMC flash memory.

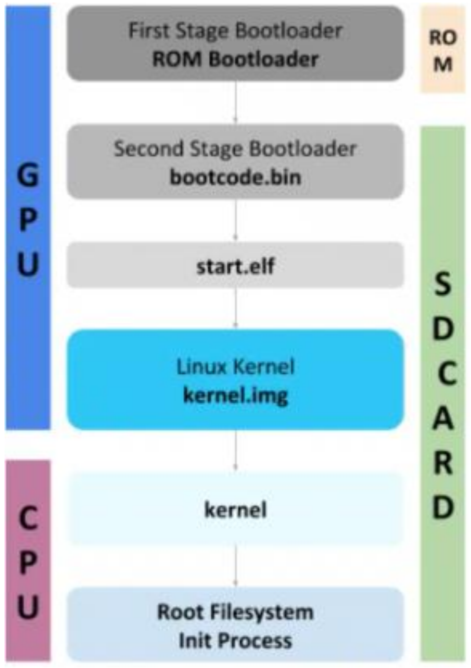
Advantages of MicroSD cards:
MicroSD cards offer several advantages over internal storage options like eMMC flash memory for Raspberry Pi boards. First off, they're more affordable, allowing users to choose storage capacities to fit their needs and budget. Second, they also offer easy updates and scalability, with capacities ranging from gigabytes to hundreds of gigabytes. Additionally, they're plug-and-play devices, which makes installation easier and increases accessibility for users of all skill levels.

Boot process of Raspberry Pi:
The Raspberry Pi goes through a number of steps during bootup that initialize the hardware and load the operating system. Below is the boot process sequence: (Ghael H et al., 2020, p.3)

Step	Description
1. Power-On	The boot procedure is initiated when the Raspberry Pi is turned on.
2. Bootloader('bootcode.bin')	The initial bootloader code stored in the SoC's (System on a chip) ROM executes, performing basic hardware initialization and loading the next stage bootloader ('bootcode.bin').
3. GPU Firmware ('start.elf')	The GPU firmware ('start.elf') is loaded by the bootloader and initializes the GPU and configures hardware components.
4. Kernel ('kernel.img')	The Linux kernel ('kernel.img') is loaded into memory by the GPU firmware from the boot partition of the microSD card.
5. Root Filesystem	The bootloader mounts the root filesystem, which holds the files and directories of the operating system after loading the kernel.
6. Init Process	The init process is also known as 'systemd', which starts system services and daemons.
7. User Space	Now the Raspberry Pi is prepared for user interaction either the command-line interface or graphical user interface after startup is finished.

Figure 2

Raspberry Pi Boot Process



Adapted from argus-sec.com

3) *Follow the online Quick Start Guide (Raspberry Pi) to assemble the Raspberry Pi system and test it with the default operating system which comes with the SD card. Insert your SD card to your Ubuntu computer and investigate its partitions and contents. How many partitions in the SD card? What each partition is created for? Identify the essential files and folders on the SD card and explain their purposes.* *(290 word)*

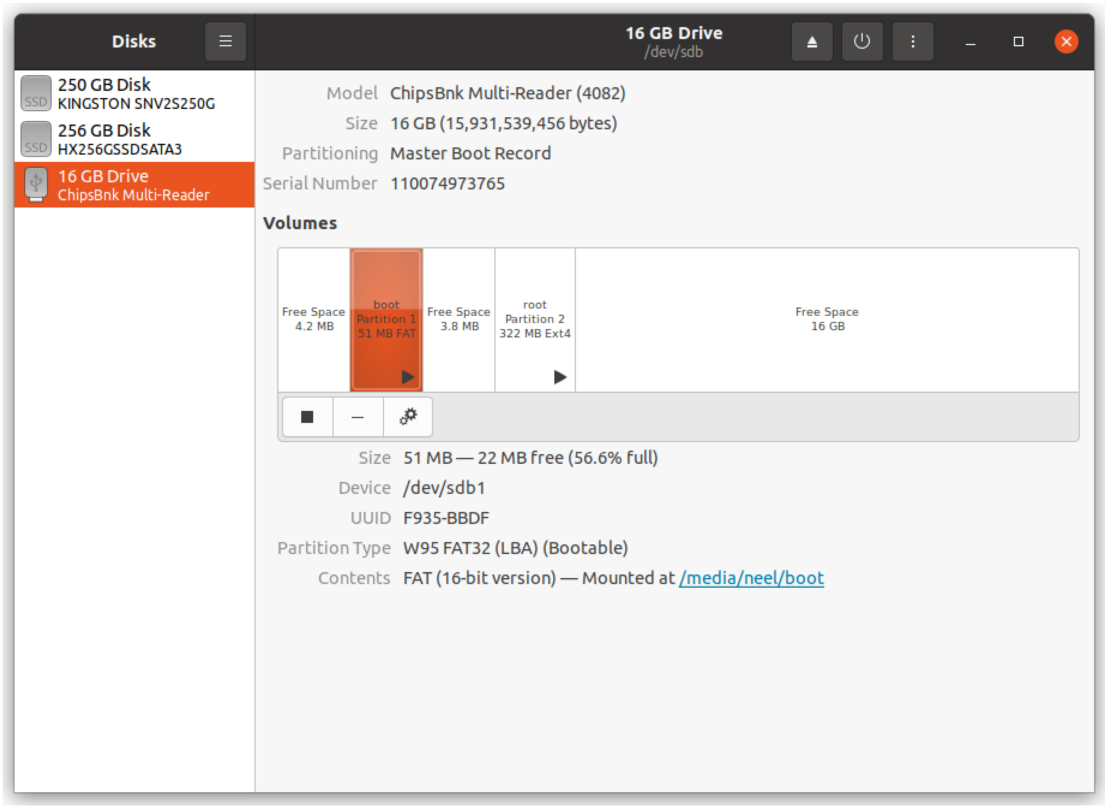
Usually a microSD card used in a Raspberry Pi has “two partitions”:

i. **Boot Partition (FAT32):**

Purpose: This is the first partition on the SD card and is formatted with a FAT filesystem. It includes all of the necessary files needed to install the Linux kernel, initialize the hardware, and set up boot parameters.

Essential Files/Folders: (Raspberry Pi, n.d.)

- bootcode.bin: The initial bootloader code that is executed by the SoC's (System on a chip) ROM.
- start.elf: GPU firmware handles the process of setting up hardware components and initializing the GPU.
- kernel.img: The Linux kernel starts the operating system and controls system resources.
- config.txt: Configuration parameters such as display settings, device interfaces, and overclocking can be customized using the config.txt file.
- cmdline.txt: The Linux kernel receives command line parameters from cmdline.txt during booting.
- Overlays(*.dtbo files): These Device Tree Overlay files configure additional hardware features or peripherals that are not enabled by default.
  etc.



Model ChipsBnk Multi-Reader (4082)
Size 16 GB (15,931,539,456 bytes)
Partitioning Master Boot Record
Serial Number 110074973765

Size 51 MB — 22 MB free (56.6% full)
Device /dev/sdb1
UUID F935-BBDF
Partition Type W95 FAT32 (LBA) (Bootable)
Contents FAT (16-bit version) — Mounted at /media/neel/boot

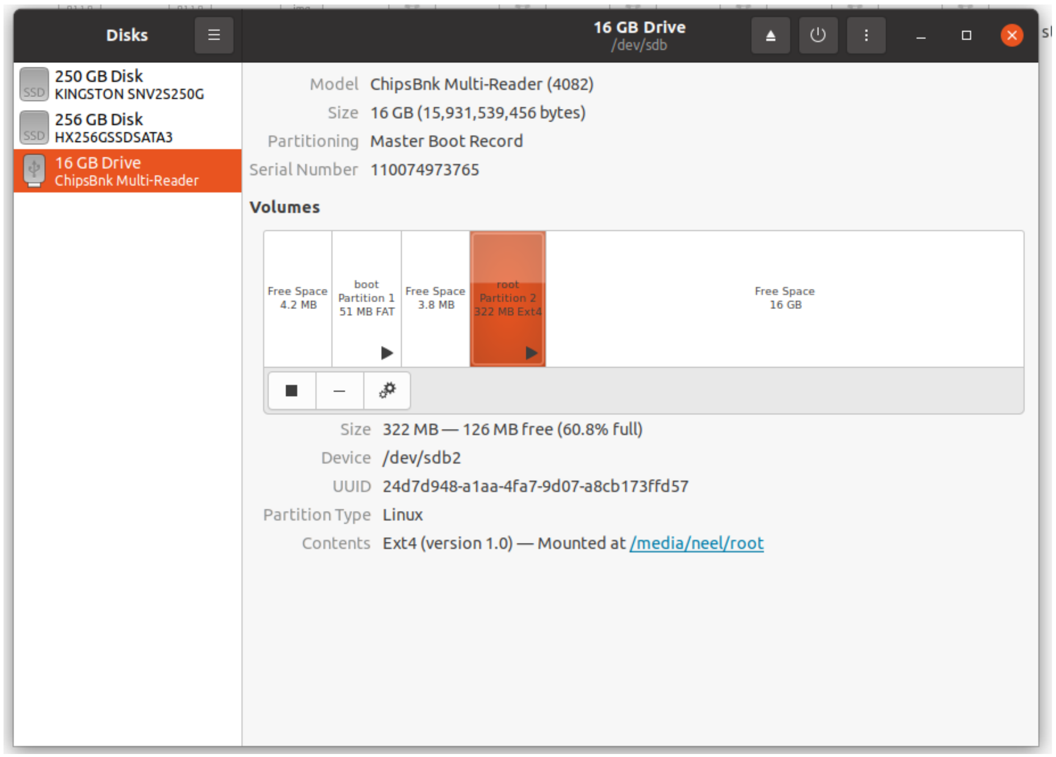
(Figure 3 - Boot partition in Linux)

ii. **Root Filesystem Partition (EXT4 or other filesystem):**

Purpose: The second partition on the SD card is formatted with a Linux filesystem (such as ext4) and contains the root filesystem of the operating system. This partition stores all the files and directories necessary for the operating system to function, including system binaries, libraries, configuration files, and user data.

Essential Files/Folders:

- /etc: This directory contains system-wide configuration files for various software components and services.
- /bin, /sbin, /usr/bin, /usr/sbin: This directory contains important system binaries and commands for system administration.
- /lib, /lib64: This directory contains shared libraries that are needed by system binaries and applications.
- /home: Default directory for user home directories.
- /var: This directory contains variable data files, such as log files, temporary files, and spool directories.
- /boot: On some configurations, a copy of the boot partition is mounted here.
  etc.



Disks

16 GB Drive
/dev/sdb

250 GB Disk
KINGSTON SNV2S250G

256 GB Disk
HX256GSSDSATA3

16 GB Drive
ChipsBnk Multi-Reader

Model ChipsBnk Multi-Reader (4082)
Size 16 GB (15,931,539,456 bytes)
Partitioning Master Boot Record
Serial Number 110074973765

Volumes

Size 322 MB — 126 MB free (60.8% full)
Device /dev/sdb2
UUID 24d7d948-a1aa-4fa7-9d07-a8cb173ffd57
Partition Type Linux
Contents Ext4 (version 1.0) — Mounted at /media/neel/root

(Figure 4 - Root partition in Linux)

4) Criticize the embedded operating systems available for Raspberry Pi. Explain the principle of cross-platform development toolchain for embedded development and its importance.

Criticize the embedded operating systems available for Raspberry Pi:

Numerous embedded operating systems are available for Raspberry Pi, although this diversity may bring downsides. Each operating system has advantages and disadvantages of its own, and beginners may find the vast number of options overwhelming. Lightweight Linux distributions could be easier to use but have less functionality than full-feature ones, which offer flexibility but need more study. Less well-liked options may receive fewer security updates, and operating systems with high resource requirements may cause the Pi's processing power to lag.

So depending on the particular requirements of your project, you should select an OS based on factors like real-time performance, customization, ease of use, or application-specific requirements like video streaming.

Principle of cross- platform development toolchain for embedded development:

Cross-platform development refers to the process of developing software that can be installed and used on several hardware or software platforms without requiring numerous changes. (VARTEQ Inc., 2023)

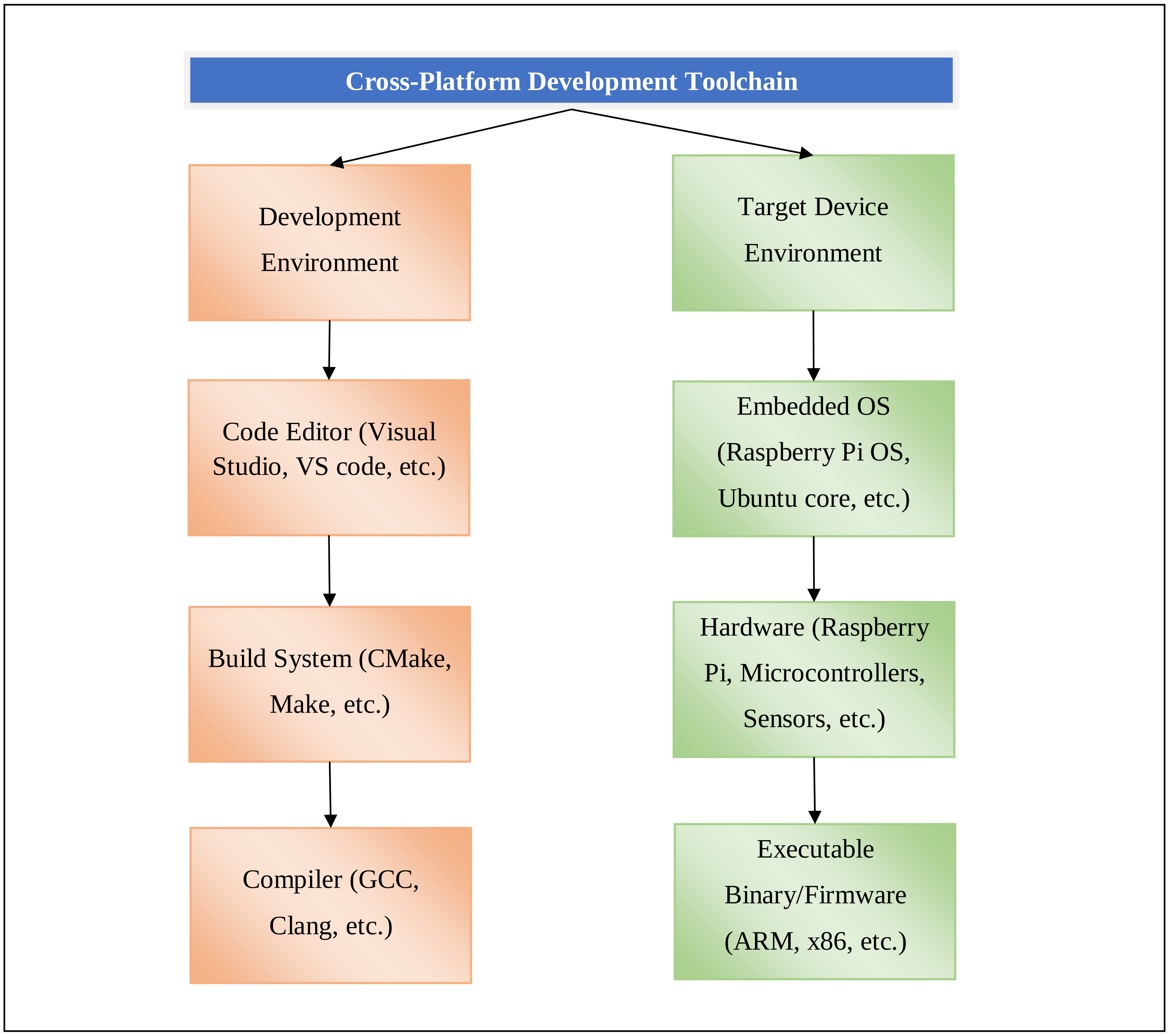
Cross-platform toolchains enable developers to write code on one platform like a PC and have it compiled so it can run on another such as an embedded device like the Raspberry Pi. However, it’s important to understand that embedded systems frequently have a variety of scenarios and architectures.

Importance of cross- platform development toolchain for embedded development:

Code Reusability: You can reuse a significant amount of your codebase on many embedded devices by using cross-platform programming. This improves overall development and minimizes the possibility of making mistakes while coding. (VARTEQ Inc., 2023)

Flexibility: Cross-platform toolchains allow developers to select the best development environment for their needs, whether it's a lightweight text editor on a remote server or an efficient IDE on a desktop computer.

Compatibility: Cross-platform toolchain ensures compatibility with various build systems, libraries, and dependencies, allowing developing to use code and resources efficiently.


Cross-Platform Development Toolchain
Development Environment
Target Device Environment
Code Editor (Visual Studio, VS code, etc.)
Embedded OS (Raspberry Pi OS, Ubuntu core, etc.)
Build System (CMake, Make, etc.)
Hardware (Raspberry Pi, Microcontrollers, Sensors, etc.)
Compiler (GCC, Clang, etc.)
Executable Binary/Firmware (ARM, x86, etc.)

(Figure 5 - Diagram of Cross-Platform Development Toolchain)

5) Read and summarize the information about Yocto. Explain the terms Poky, oe-core, recipe, layer and BitBake within the context of Yocto? Elaborate on Yocto development workflow to the largest extent.

Yocto Project:
Yocto is an open-source project that offers tools and templates for creating custom Linux distributions for embedded systems (Mankodi P et al., 2017, p.1). Its goal is to make embedded Linux distribution development and maintenance easier by offering a framework that is adaptable and customizable. The Yocto Project has developed over more than ten years to become one of the biggest communities within the Linux Foundation's network of cooperative open-source projects.

<u>Benefits of the Yocto Project</u>:

- Improves cooperation and reusability between open-source and commercial communities.
- Minimizes effort duplication, and utilizes the advantages of other open-source initiatives, such as Eclipse and OpenEmbedded.

Poky:
Yocto's reference distribution is called Poky. It acts as a foundation for developing unique embedded Linux distributions. The BitBake build tool, metadata for creating common packages, and the OpenEmbedded-Core (oe-core) layer are all included in Poky.

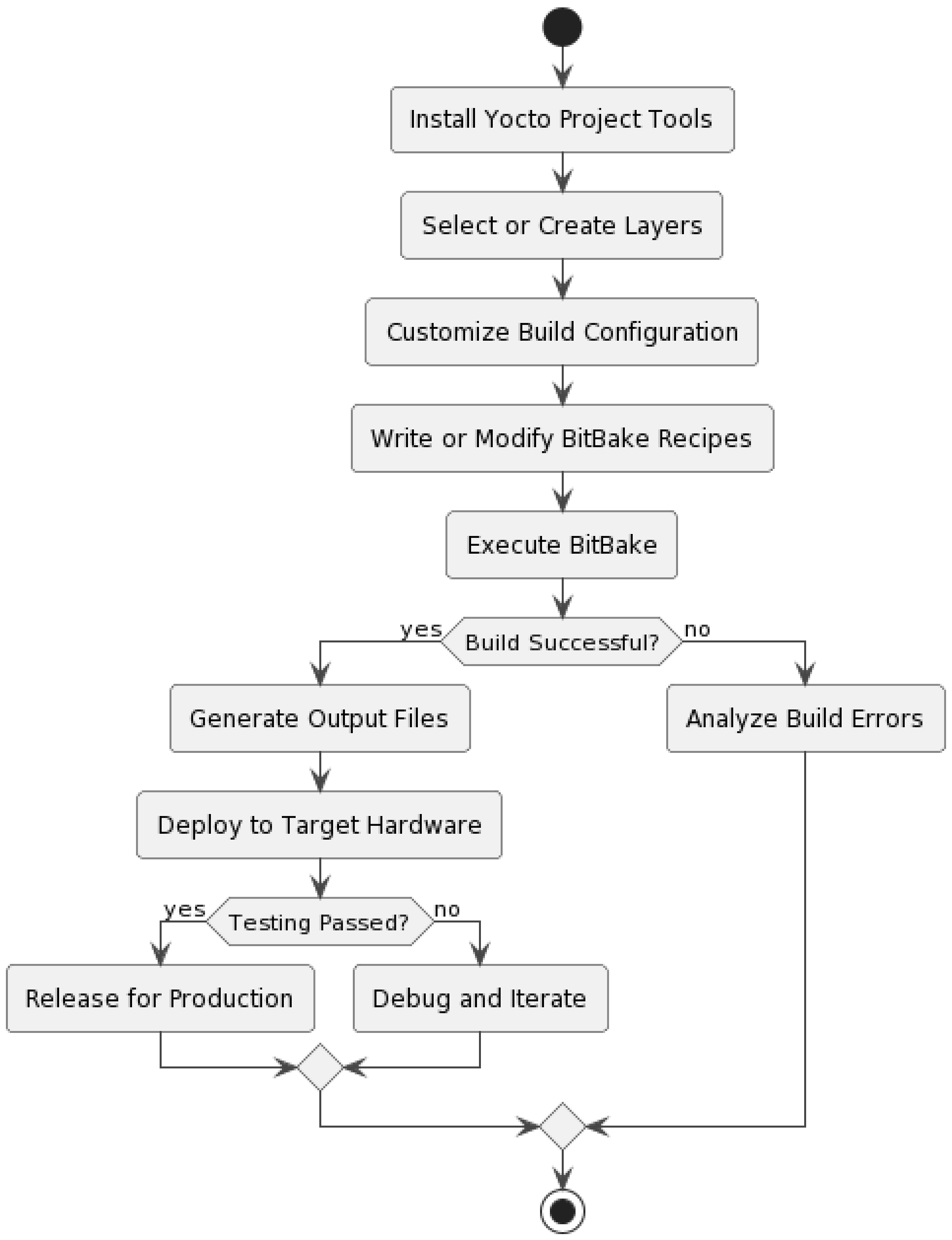
OE-Core (OpenEmbedded-Core):
Within the Yocto Project, there is a layer called OpenEmbedded-Core that offers build scripts, configuration files, and necessary recipes for assembling a minimal Linux distribution. It contains essential parts for a working system, such as the Linux kernel, bootloader, libraries, and utilities.

Recipe:
A collection of metadata files that explain how to compile a software package from source code is referred as a recipe in the Yocto. Usually, recipes include steps for downloading the source code, setting up build parameters, installing fixes, generating the code, and packaging the final binaries. Layers are used to organize recipes, which are written in the BitBake syntax.

Layer:
A layer is a collection of configuration files, recipes, and other metadata that expands the Yocto Project's capabilities. Layers let developers enhance their embedded Linux editions with new features, packages, and customizations.

BitBake:
It downloads source code, reads metadata files, resolves dependencies, carries out build operations, and produces output files like binary packages and filesystem images. It makes use of a collection of task descriptions and dependency graphs to guarantee that the build process is effective and predictable.

Yocto Development Workflow:



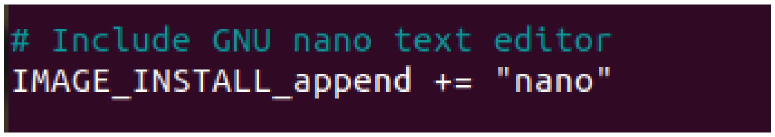
(Figure 6 - Flowchart of Yocto Development Workflow)

<u>Install Yocto Project Tools</u>: Set up Poky, BitBake, and dependencies.
<u>Select/Create Layers</u>: Select or generate layers using recipes and parameters.
<u>Customize Build Configuration</u>: Modify the `local.conf` and `bblayers.conf` files to provide appropriate settings for the target.
<u>Write/Modify Recipes</u>: Define dependencies and build instructions.
<u>Execute BitBake</u>: Run BitBake to build the project.
<u>Generate Output</u>: Get images and binary packages.
<u>Deploy/Test</u>: Deploy to target hardware and test functionality.
<u>Iterate/Debug</u>: Refine configurations and address issues.
<u>Release</u>: Prepare documentation and release for production.

Part 2: Embedded System Design

1) Develop and customize a Yocto-based embedded Linux system for the Raspberry Pi 3 single-board computer. Your image should include the GNU nano text editor by default. Furthermore, the Dropbear SSH server and the splash screen during the booting process should be incorporated as the image extra features. Moreover, the Linux kernel module for the USB devices should be configured so that it is automatically loaded into the Linux kernel at start-up.

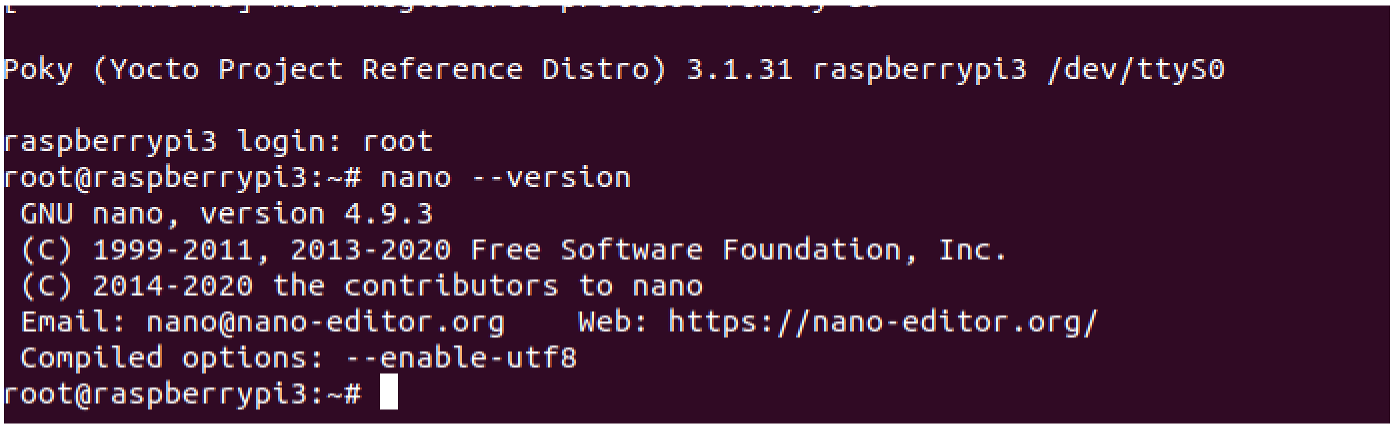
GNU nano text editor:



```
# Include GNU nano text editor
IMAGE_INSTALL_append += "nano"
```

(Figure 7)

As you can see in the above figure, I've included this line in my local configuration file (local.conf`) inside my build directory to include GNU nano in the image.



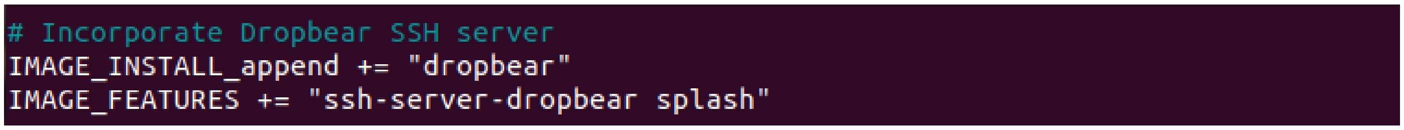
```
Poky (Yocto Project Reference Distro) 3.1.31 raspberrypi3 /dev/ttyS0

raspberrypi3 login: root
root@raspberrypi3:~# nano --version
 GNU nano, version 4.9.3
 (C) 1999-2011, 2013-2020 Free Software Foundation, Inc.
 (C) 2014-2020 the contributors to nano
 Email: nano@nano-editor.org    Web: https://nano-editor.org/
 Compiled options: --enable-utf8
root@raspberrypi3:~# 
```

(Figure 8)

After successfully building and flashing the Yocto-based image into the SD card, I verified the implementation of the GNU nano editor by accessing the command-line interface of the Raspberry Pi. First, I used the command `nano --version` to see if nano was installed. The above figure confirmed the GNU nano editor's existence in the system by displaying its version information.

I then typed nano into the terminal to open the Nano editor. I was able to create, edit, and view text files straight from the command line after this unlocked the GNU nano interface.

Dropbear SSH server:
Changes were made to the local configuration file (local.conf) in order to integrate the Dropbear SSH server into the Yocto-based embedded Linux system for the Raspberry Pi 3. The below image lines were added to ensure that the Dropbear SSH server is included in the image.

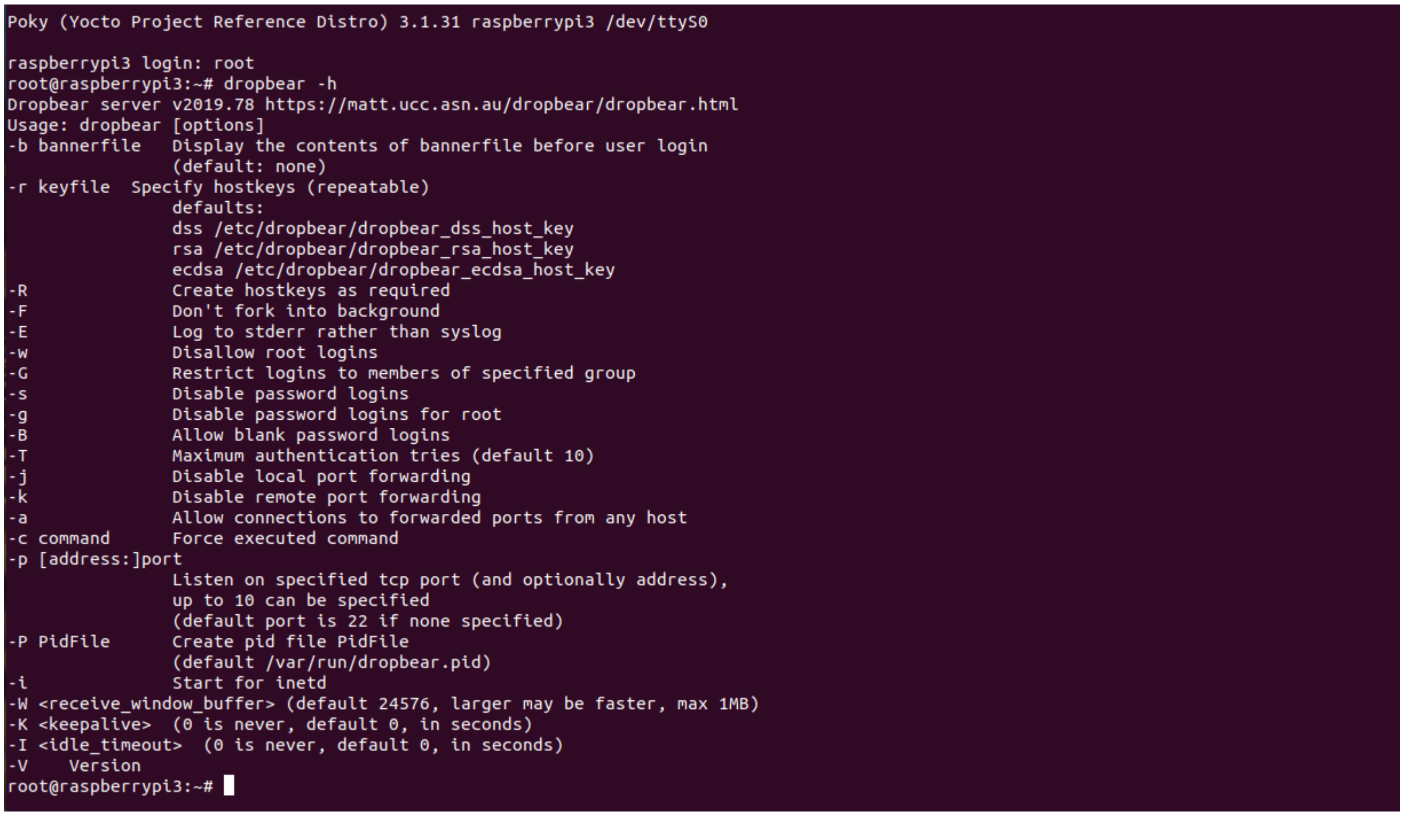


```
# Incorporate Dropbear SSH server
IMAGE_INSTALL_append += "dropbear"
IMAGE_FEATURES += "ssh-server-dropbear splash"
```

(Figure 9)

IMAGE_INSTALL_append += "dropbear": The Dropbear package is added to the list of packages that need to be installed in the image by this line. Dropbear is a lightweight SSH server and client implementation suitable for resource-constrained environments.

IMAGE_FEATURES += "ssh-server-dropbear": The ssh-server-dropbear feature is added to the image by this line. This feature makes sure that the image has the Dropbear SSH server component enabled.



```
Poky (Yocto Project Reference Distro) 3.1.31 raspberrypi3 /dev/ttyS0

raspberrypi3 login: root
root@raspberrypi3:~# dropbear -h
Dropbear server v2019.78 https://matt.ucc.asn.au/dropbear/dropbear.html
Usage: dropbear [options]
-b bannerfile   Display the contents of bannerfile before user login
                (default: none)
-r keyfile  Specify hostkeys (repeatable)
                defaults:
                dss /etc/dropbear/dropbear_dss_host_key
                rsa /etc/dropbear/dropbear_rsa_host_key
                ecdsa /etc/dropbear/dropbear_ecdsa_host_key
-R              Create hostkeys as required
-F              Don't fork into background
-E              Log to stderr rather than syslog
-w              Disallow root logins
-G              Restrict logins to members of specified group
-s              Disable password logins
-g              Disable password logins for root
-B              Allow blank password logins
-T              Maximum authentication tries (default 10)
-j              Disable local port forwarding
-k              Disable remote port forwarding
-a              Allow connections to forwarded ports from any host
-c command      Force executed command
-p [address:]port
                Listen on specified tcp port (and optionally address),
                up to 10 can be specified
                (default port is 22 if none specified)
-P PidFile      Create pid file PidFile
                (default /var/run/dropbear.pid)
-i              Start for inetd
-W <receive_window_buffer> (default 24576, larger may be faster, max 1MB)
-K <keepalive>  (0 is never, default 0, in seconds)
-I <idle_timeout>  (0 is never, default 0, in seconds)
-V    Version
root@raspberrypi3:~# █
```

(Figure 10)

From the above figure, we can see that the `dropbear -h` command displayed the help information for the Dropbear SSH server, verifying its existence within the system. Therefore, it indicates that the SSH server component has been added to the Yocto-based embedded Linux system for the Raspberry Pi 3. By offering secure remote access via SSH, Dropbear SSH server enhances the functionality of the system and makes necessary activities like remote administration and debugging easier to accomplish.

2) Develop Linux support (a loadable kernel module) for HC-SR04 sensor that makes use of both VFS API and sysfs file system.

a. Compose a loadable kernel module will use the VFS API as follows:

i. The write() system call will trigger the sensor, and it will measure the echo pulse duration,

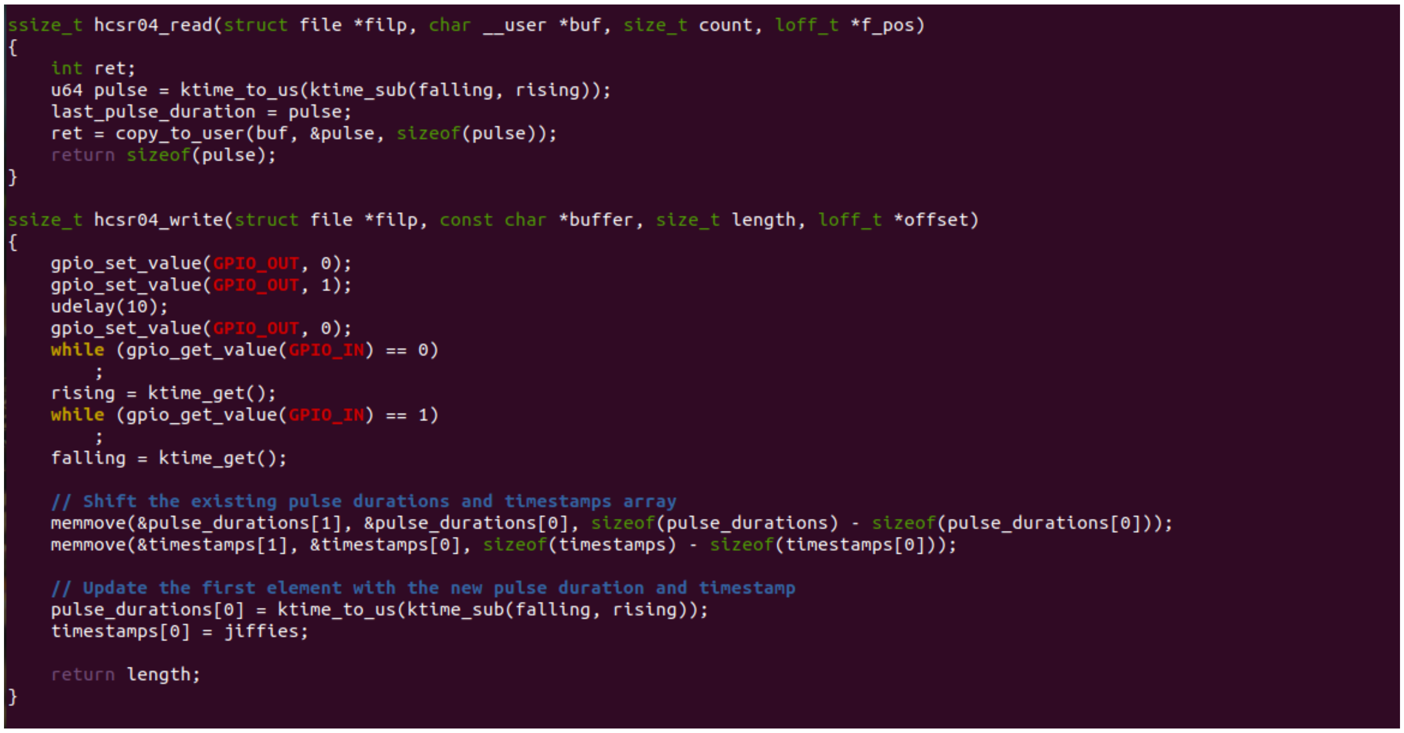
ii. The read() system call will provide to the user level the measured echo pulse duration in microseconds.

b. The same loadable Kernel module will use the sysfs to record and provide the user individually with:

i. the last echo pulse duration read, and the distance measured from the sensor with timestamps,

ii. the last 5 echo pulse duration read, and the distance measured from the sensor with timestamps.

VFS Usage Functions:



```
ssize_t hcsr04_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
    int ret;
    u64 pulse = ktime_to_us(ktime_sub(falling, rising));
    last_pulse_duration = pulse;
    ret = copy_to_user(buf, &pulse, sizeof(pulse));
    return sizeof(pulse);
}

ssize_t hcsr04_write(struct file *filp, const char *buffer, size_t length, loff_t *offset)
{
    gpio_set_value(GPIO_OUT, 0);
    gpio_set_value(GPIO_OUT, 1);
    udelay(10);
    gpio_set_value(GPIO_OUT, 0);
    while (gpio_get_value(GPIO_IN) == 0)
        ;
    rising = ktime_get();
    while (gpio_get_value(GPIO_IN) == 1)
        ;
    falling = ktime_get();

    // Shift the existing pulse durations and timestamps array
    memmove(&pulse_durations[1], &pulse_durations[0], sizeof(pulse_durations) - sizeof(pulse_durations[0]));
    memmove(&timestamps[1], &timestamps[0], sizeof(timestamps) - sizeof(timestamps[0]));

    // Update the first element with the new pulse duration and timestamp
    pulse_durations[0] = ktime_to_us(ktime_sub(falling, rising));
    timestamps[0] = jiffies;

    return length;
}
```

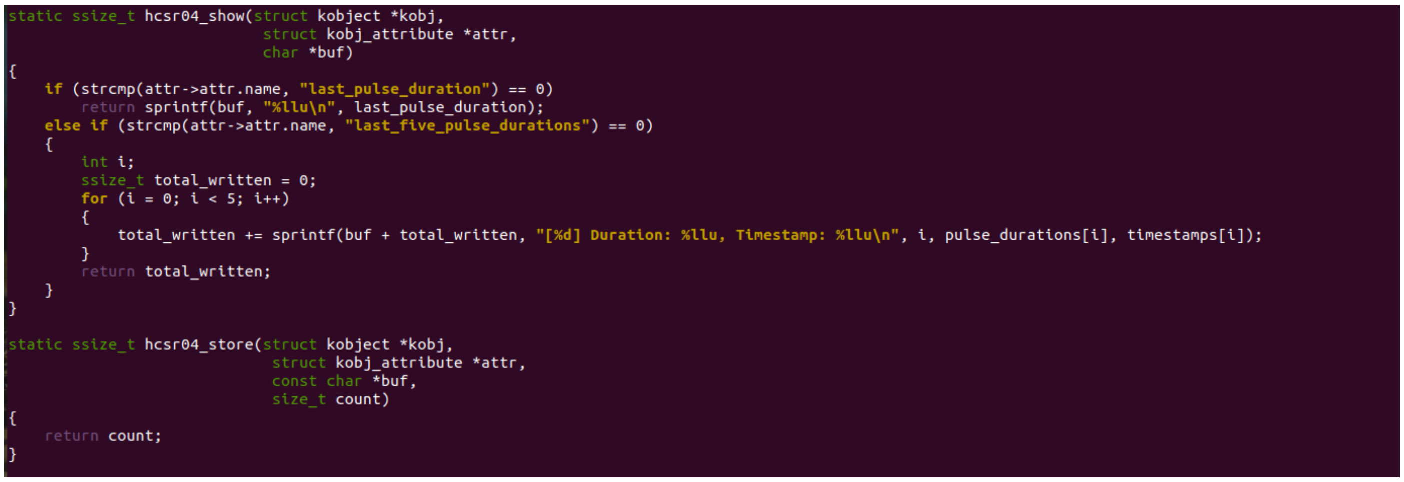
(Figure 11 - read & write function inkernel module)

VFS usage functions in the kernel module, such as hcsr04_write() and hcsr04_read(), facilitate communication between user-space programs and the kernel by handling write and read operations respectively on the device file associated with the module. These operations activate the sensor, detect the duration of echo pulses, and give user-space apps access to this information.

Write function (hcsr04_write): The hcsr04 write function initiates the HC-SR04 sensor, records the echo pulse timestamps, computes the pulse duration, and saves it in the last_pulse_duration variable. It also modifies arrays that hold the timestamps and pulse lengths of the previous five pulses. This feature lets user-space programs start using the sensor to measure distance.

Read function (hcsr04_read): The most recent pulse duration recorded by the HC-SR04 sensor is obtained from the last_pulse_duration variable via the hcsr04_read function. The echo pulse return time is then represented by this duration, which is copied into the user-space buffer. With the use of this feature, user-space programmes can get real-time data regarding the sensor's measured distance.

Sysfs Usage Functions:



```c
static ssize_t hcsr04_show(struct kobject *kobj,
                           struct kobj_attribute *attr,
                           char *buf)
{
    if (strcmp(attr->attr.name, "last_pulse_duration") == 0)
        return sprintf(buf, "%llu\n", last_pulse_duration);
    else if (strcmp(attr->attr.name, "last_five_pulse_durations") == 0)
    {
        int i;
        ssize_t total_written = 0;
        for (i = 0; i < 5; i++)
        {
            total_written += sprintf(buf + total_written, "[%d] Duration: %llu, Timestamp: %llu\n", i, pulse_durations[i], timestamps[i]);
        }
        return total_written;
    }
}

static ssize_t hcsr04_store(struct kobject *kobj,
                            struct kobj_attribute *attr,
                            const char *buf,
                            size_t count)
{
    return count;
}
```

(Figure 12 - show & store fuction in kernel module)

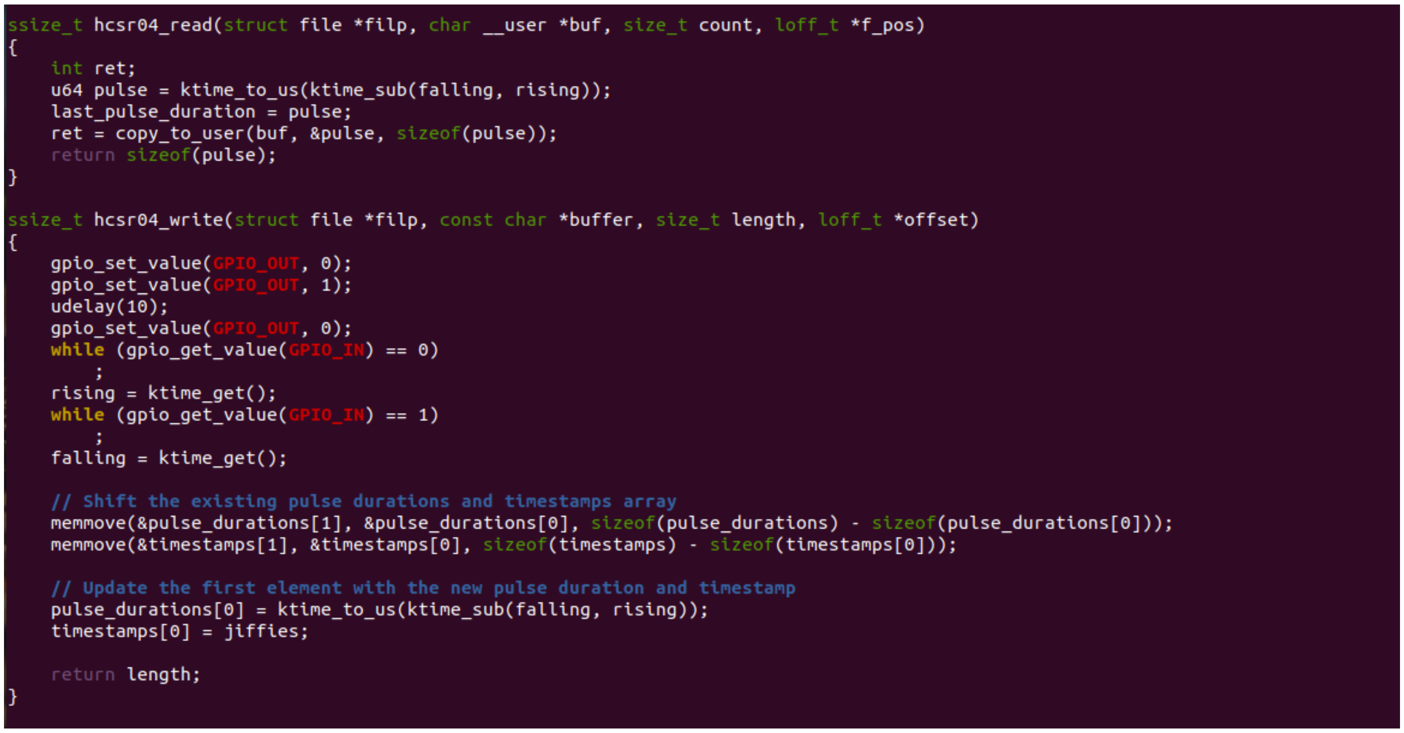
Sysfs usage functions like hcsr04_show() and hcsr04_store(), control how user-space programs communicate with the kernel module via the sysfs filesystem. These routines allow the HC-SR04 sensor module to be monitored and controlled from user space by presenting sensor data, including timestamps and echo pulse durations, to users via sysfs attributes.

<u>Show function (hcsr04_show)</u>: Through the Sysfs interface, the hcsr04_show function offers user-readable details on the HC-SR04 sensor module. When users query it, it formats and displays data like timestamps and pulse lengths. Users can obtain essential information regarding the performance and operation of the sensor with this capability.

<u>Store function (hcsr04_store)</u>: For the HC-SR04 sensor module, write operations on sysfs attributes are managed by the hcsr04_store function. Although it is very minimally implemented now, in the future it can be extended to allow particular operations for changing attribute values. The store function, though straightforward, paves the way for future improvements to the sensor module's sysfs interface.

3) Develop a module test application which would iteratively trigger the sensor by executing the "write" system call, "read" the echo pulse duration from the sensor, and finally display the duration and the corresponding distance with timestamps as produced by the kernel module. The number of times the application program iterates should be dictated by the user inputs at the command-line console.

kernel module part:

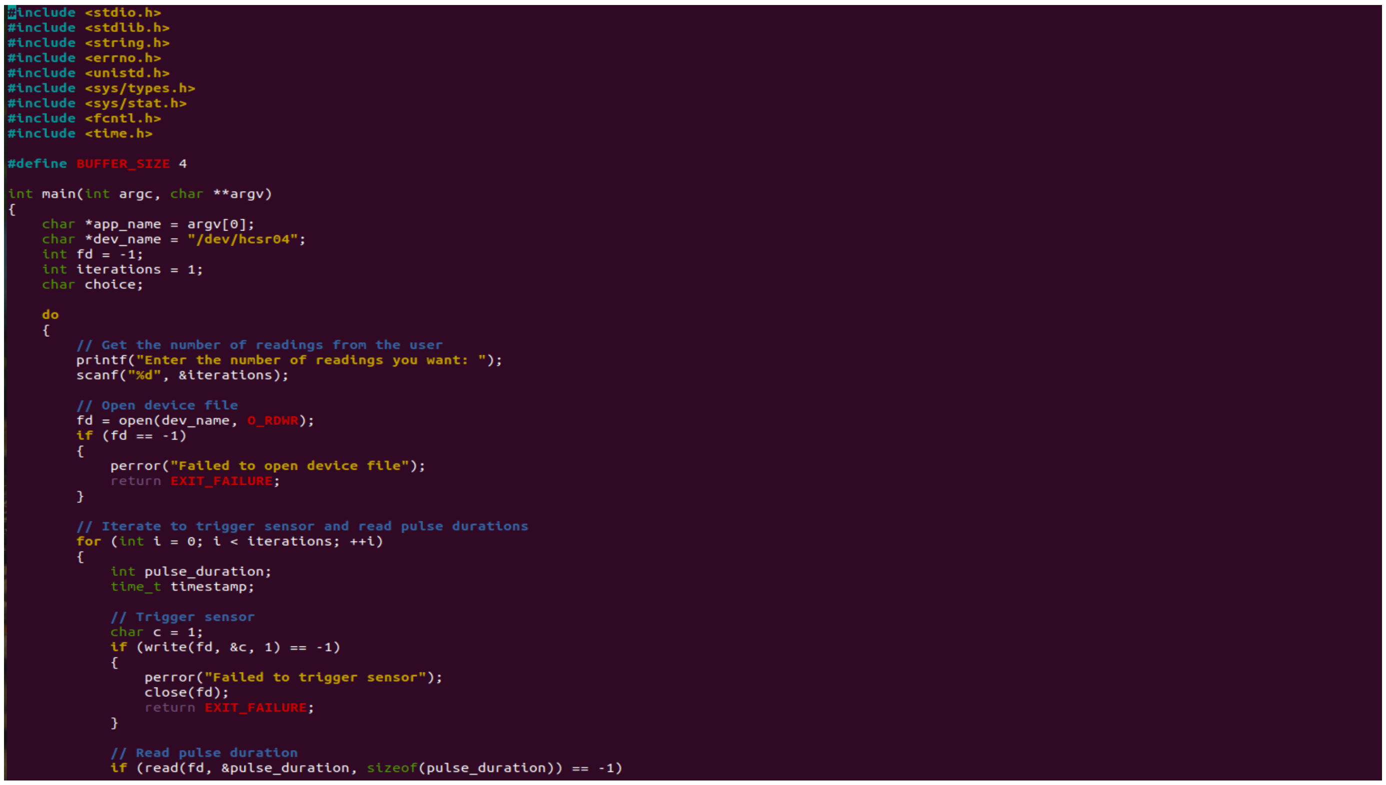


```c
ssize_t hcsr04_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
    int ret;
    u64 pulse = ktime_to_us(ktime_sub(falling, rising));
    last_pulse_duration = pulse;
    ret = copy_to_user(buf, &pulse, sizeof(pulse));
    return sizeof(pulse);
}

ssize_t hcsr04_write(struct file *filp, const char *buffer, size_t length, loff_t *offset)
{
    gpio_set_value(GPIO_OUT, 0);
    gpio_set_value(GPIO_OUT, 1);
    udelay(10);
    gpio_set_value(GPIO_OUT, 0);
    while (gpio_get_value(GPIO_IN) == 0)
        ;
    rising = ktime_get();
    while (gpio_get_value(GPIO_IN) == 1)
        ;
    falling = ktime_get();

    // Shift the existing pulse durations and timestamps array
    memmove(&pulse_durations[1], &pulse_durations[0], sizeof(pulse_durations) - sizeof(pulse_durations[0]));
    memmove(&timestamps[1], &timestamps[0], sizeof(timestamps) - sizeof(timestamps[0]));

    // Update the first element with the new pulse duration and timestamp
    pulse_durations[0] = ktime_to_us(ktime_sub(falling, rising));
    timestamps[0] = jiffies;

    return length;
}
```

(Figure 13)

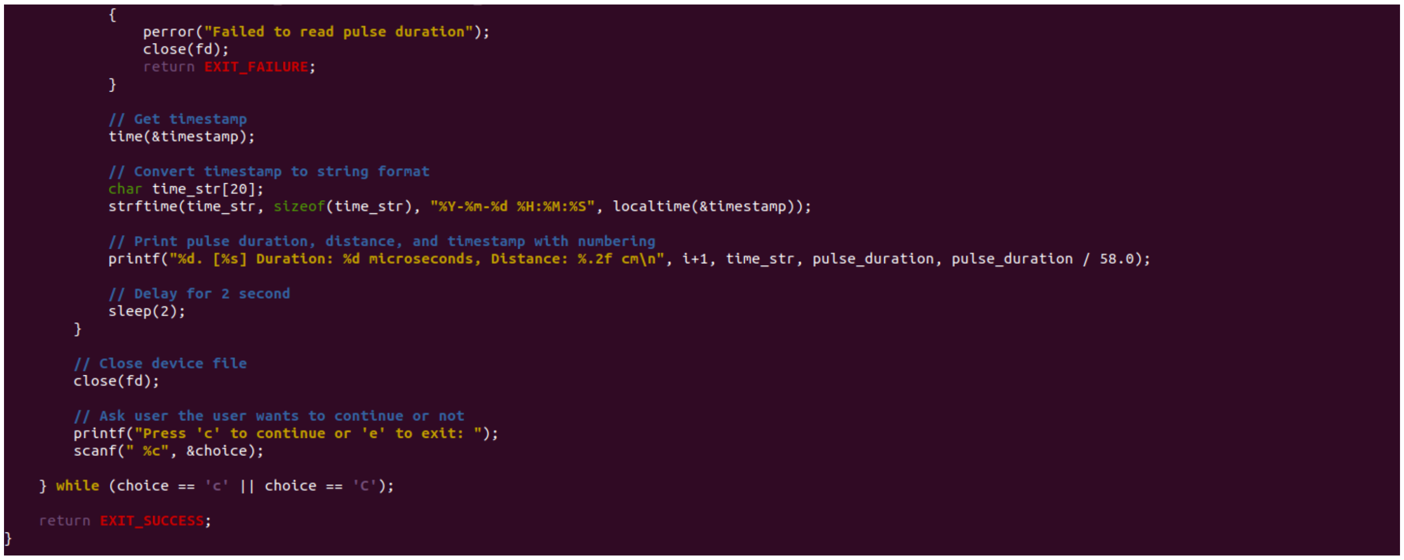
The read and write functions are shown in the above figure, which show the fundamental functions of the kernel module. These functions are the main parts in charge of interacting with the HC-SR04 sensor and enabling data transfer between the user and kernel spaces.

Test application part:



```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <time.h>

#define BUFFER_SIZE 4

int main(int argc, char **argv)
{
    char *app_name = argv[0];
    char *dev_name = "/dev/hcsr04";
    int fd = -1;
    int iterations = 1;
    char choice;

    do
    {
        // Get the number of readings from the user
        printf("Enter the number of readings you want: ");
        scanf("%d", &iterations);

        // Open device file
        fd = open(dev_name, O_RDWR);
        if (fd == -1)
        {
            perror("Failed to open device file");
            return EXIT_FAILURE;
        }

        // Iterate to trigger sensor and read pulse durations
        for (int i = 0; i < iterations; ++i)
        {
            int pulse_duration;
            time_t timestamp;

            // Trigger sensor
            char c = 1;
            if (write(fd, &c, 1) == -1)
            {
                perror("Failed to trigger sensor");
                close(fd);
                return EXIT_FAILURE;
            }

            // Read pulse duration
            if (read(fd, &pulse_duration, sizeof(pulse_duration)) == -1)
```

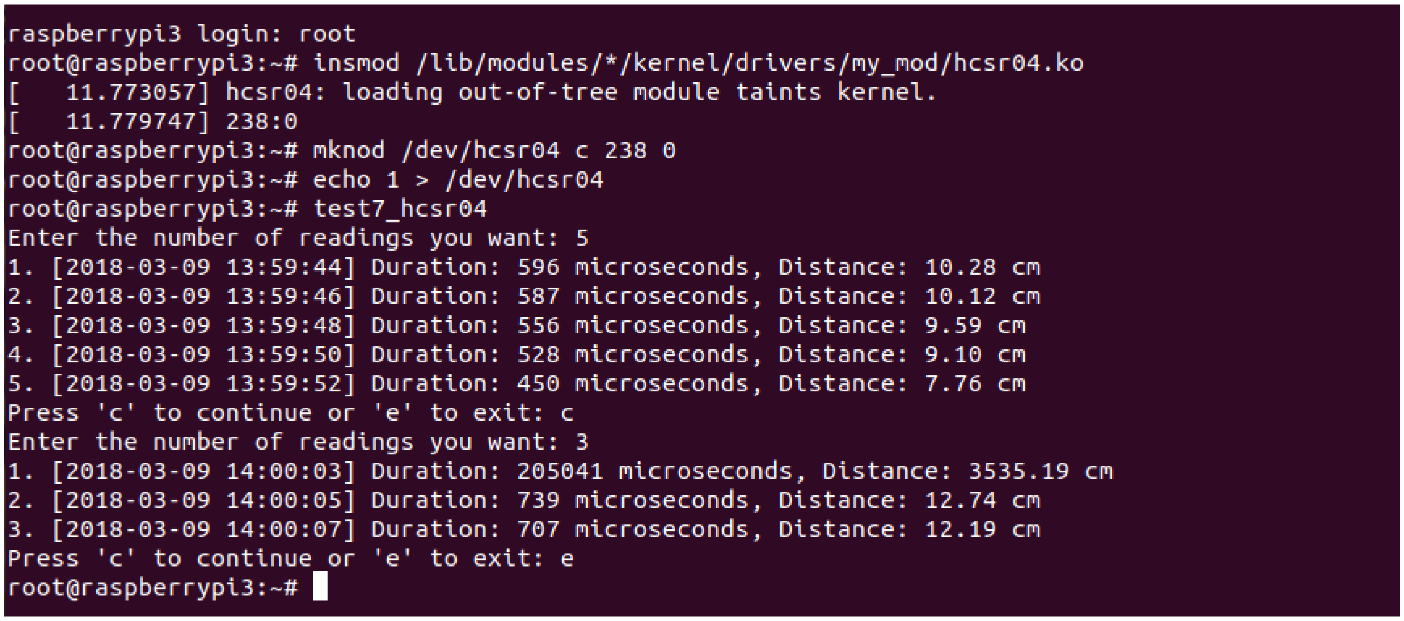
(Figure 14 - Test code 1)



```
            {
                perror("Failed to read pulse duration");
                close(fd);
                return EXIT_FAILURE;
            }

            // Get timestamp
            time(&timestamp);

            // Convert timestamp to string format
            char time_str[20];
            strftime(time_str, sizeof(time_str), "%Y-%m-%d %H:%M:%S", localtime(&timestamp));

            // Print pulse duration, distance, and timestamp with numbering
            printf("%d. [%s] Duration: %d microseconds, Distance: %.2f cm\n", i+1, time_str, pulse_duration, pulse_duration / 58.0);

            // Delay for 2 second
            sleep(2);
        }

        // Close device file
        close(fd);

        // Ask user the user wants to continue or not
        printf("Press 'c' to continue or 'e' to exit: ");
        scanf(" %c", &choice);

    } while (choice == 'c' || choice == 'C');

    return EXIT_SUCCESS;
}
```

(Figure 15 - Test code 2)

The test code acts as a mediator between the user and the HC-SR04 sensor, enabling the assessment of its performance. Users determine how many readings they want, which determines how many times the program runs. When the program runs, it opens the device file for the sensor and starts the data-gathering loop. The sensor is activated, the pulse length is measured, and timestamps are logged during each repetition. The user is shown the resultant data, which includes pulse duration converted to distance. Following an iteration, the device file is closed, and users are presented with an option to proceed or exit.



```
raspberrypi3 login: root
root@raspberrypi3:~# insmod /lib/modules/*/kernel/drivers/my_mod/hcsr04.ko
[   11.773057] hcsr04: loading out-of-tree module taints kernel.
[   11.779747] 238:0
root@raspberrypi3:~# mknod /dev/hcsr04 c 238 0
root@raspberrypi3:~# echo 1 > /dev/hcsr04
root@raspberrypi3:~# test7_hcsr04
Enter the number of readings you want: 5
1. [2018-03-09 13:59:44] Duration: 596 microseconds, Distance: 10.28 cm
2. [2018-03-09 13:59:46] Duration: 587 microseconds, Distance: 10.12 cm
3. [2018-03-09 13:59:48] Duration: 556 microseconds, Distance: 9.59 cm
4. [2018-03-09 13:59:50] Duration: 528 microseconds, Distance: 9.10 cm
5. [2018-03-09 13:59:52] Duration: 450 microseconds, Distance: 7.76 cm
Press 'c' to continue or 'e' to exit: c
Enter the number of readings you want: 3
1. [2018-03-09 14:00:03] Duration: 205041 microseconds, Distance: 3535.19 cm
2. [2018-03-09 14:00:05] Duration: 739 microseconds, Distance: 12.74 cm
3. [2018-03-09 14:00:07] Duration: 707 microseconds, Distance: 12.19 cm
Press 'c' to continue or 'e' to exit: e
root@raspberrypi3:~# 
```

(Figure 16)

From the above figure, we can see that I loaded the hcsr04 kernel module with insmod to start the Linux kernel's integration of the HC-SR04 sensor. In order to facilitate communication between user-space apps and the kernel module, mknod was used to construct a device node called /dev/hcsr04. One could activate the sensor by writing 1 to /dev/hcsr04.

When test7_hcsr04 was run, it asked how many readings to take, triggered the sensor repeatedly, and displayed timestamps (date & time), pulse durations (in microseconds), and distances (in cm). Users have the choice to either continue or exit the process after it is finished. Moreover, two seconds have been added to the reading interval to provide the sensor enough time to stabilize and produce accurate results.

The reason time and date are false readings is because the Raspberry Pi lacks a real-time clock (RTC) module and network time synchronization, which is the cause of the time and date disparity.

One drive link of source codes, recipes, makefiles, built image & videos:

Embedded Operating Systems Coursework

(Note: Please select 1080p for better resolution)

References

1) Raspberry pi.com. (n.d.). *Raspberry Pi Schematic*. https://www.raspberrypi.com/documentation/computers/raspberry-pi.html

2) Ghael, H.D. et al. (2020). A Review Paper on Raspberry Pi and its Applications. *International Journal of Advances in Engineering and Management (IJAEM)*, *2(12)*, 3. https://dx.doi.org/10.35629/5252-0212225227

3) Argus-sec.com. (n.d.). *Raspberry Pi Boot Process.* https://argus-sec.com/blog/engineering-blog/raspberry-pi-remote-flashing/

4) Raspberry Pi. (n.d.). *SD card boot files.* https://github.com/raspberrypi/documentation/blob/develop/documentation/asciidoc/computers/configuration/boot_folder.adoc

5) VARTEQ Inc. (2023). *Principle of cross- platform development.* https://www.linkedin.com/pulse/cross-platform-development-embedded-systems-varteq/

6) VARTEQ Inc. (2023). *Advantages of Cross-Platform Development for Embedded Systems.* https://www.linkedin.com/pulse/cross-platform-development-embedded-systems-varteq/

7) Mankodi, P. et al. (2017). Enhancement of automation testing system using Yocto project. *2017 International conference of Electronics, Communication and Aerospace Technology (ICECA)*, 1. *https://dx.doi.org/10.1109/ICECA.2017.8203630*