7041CEM Coursework Submission

Jerome Samuels-Clarke

10473050

Module Leader – **Dr Server Kasap**

March 2022

Contents

1.1 Introduction

This report is about using Yocto to build a custom embedded Linux operating system, targeted to the Raspberry Pi 3B+. A kernel module was created to control the HC-SR04 sensor, which can be controlled with through a user application, to trigger and read the sensor.

Cloud link:

10473050_7041CEM_Submission

1.2 The Raspberry PI

The Raspberry Pi (Pi) comes in many different models at a variety of prices. The Pi model used in this report is the 3B+ model, which is a single board computer (SBC) that is packed with features, such as USB 2.0 support, GPIO, HDMI output and many more. The heart of the model comes from the ARM Cortex-A53, which is a 64-bit SoC capable of running at 1.4GHz. Lastly, Networking capability is also included on the Pi with Wi-Fi, Bluetooth, and Ethernet support. The Pi needs to either use an SD card for storage or an external hard drive connected to USB. The size of the SD on the Pi can be as large as 32GB. The schematic for the 3B+ is shown in Figure 1.

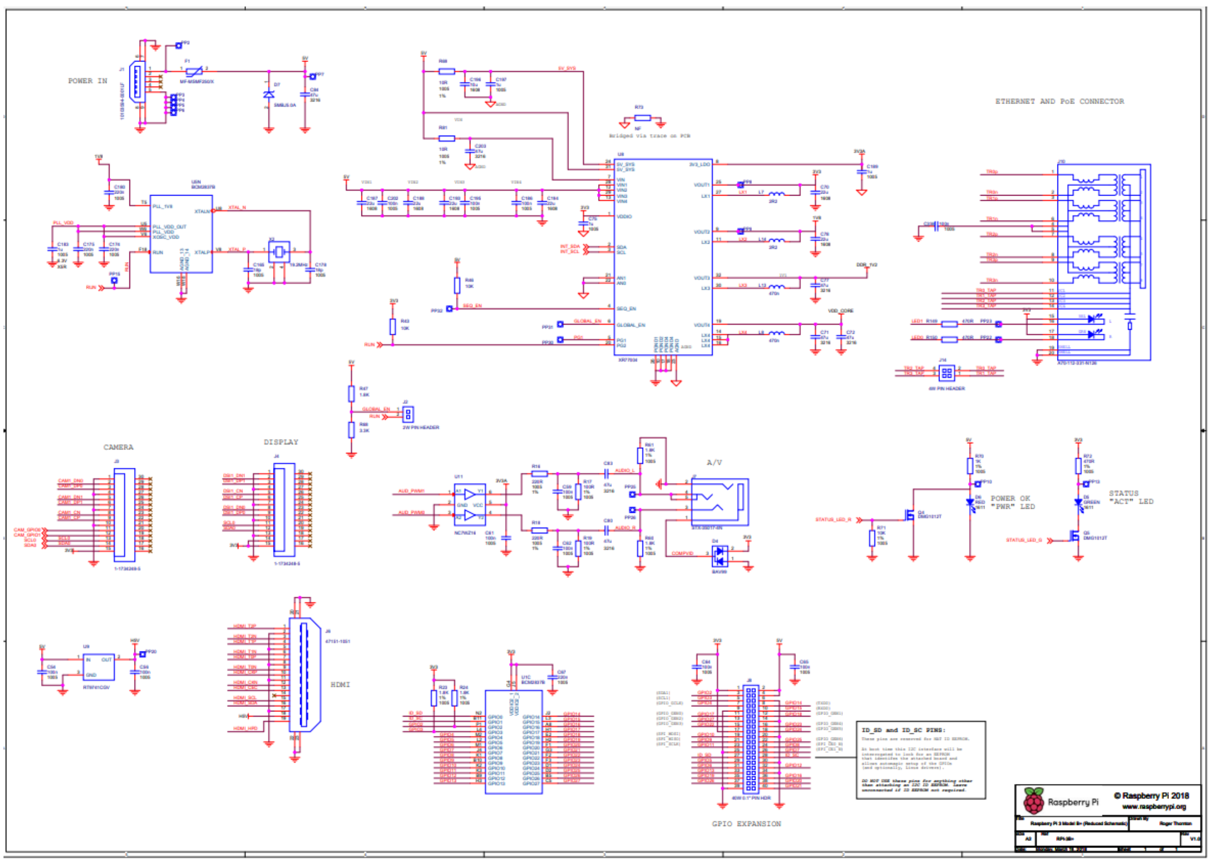


Figure 1: Schematic of Raspberry Pi 3B+. Raspberry Pi Documentation. (n.d.). Raspberrypi. Retrieved 7 March 2022, from https://www.raspberrypi.com/documentation/computers/raspberry-pi.html

The boot process of the Pi can be summarised in a few steps, however in each of these steps many instructions are beings executed to configure the Pi:

1. The first step in the boot process of the Pi is to execute the first stage bootloader. This is located on the Pi chip, which used to mount the SD card to run the second stage bootloader.
2. The second stage bootloader will set up the SDRAM from the bootcode.bin file. After the start.elf file will be loaded to initialise the hardware peripherals, as well as load the kernel image.
3. Lastly, during the kernel loading, many different parts of the system will be configured. The *init()* function will also run, which will start the system initialisation.

1.2.1 Building a Raspberry Pi OS

The default operating system for the Pi, is called the Raspberry Pi OS (Figure 2). The OS image can be downloaded directly from the website, can be setup in a matter of minutes. The basic steps for doing this is:

1. Download the Raspberry PI OS image.
2. Flash the image onto an SD card and insert it into the Pi.
3. Connect a keyboard, mouse, HDMI cable and power on the Pi.

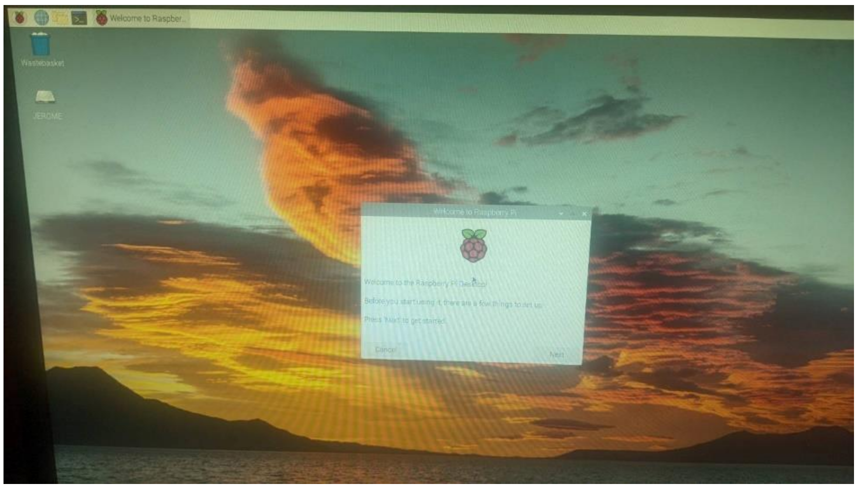


Figure 2: Boot screen of the Raspberry Pi OS

The boot partition has everything needed to boot the Pi and contains configuration files. The rootfs partition is created to hold the filesystem, located in the root directory. The rootfs contains many directories and subdirectories that contain configuration files, Makefiles, libraries etc (Hallinan, 2010, p. 134). A summary of each of the directories is provided in Table 1.

Table 1: Summary of directories found in the rootfs.

Directory	Use
bin	User binaries
dev	Device files, represented as files
etc	Configuration and startup files
home	Home directory for user(s)
lib	Contains libraries
mnt	Mount location for user devices
opt	Optional files
proc	Contains information about the system
root	Root user directory
sbin	System binary files
tmp	Stores temporary files
usr	Read-only user files
var	Variable files e.g. system logs

1.2.2 Raspberry Pi OS vs Customised Operating System

Although the standard operating system for Pi is popular due to its quick development to boot and simplicity, developing a customised operating system brings many advantages. The Raspberry Pi OS image when installed is around 4GB. In comparison custom OS can be a lot smaller depending on the packages installed. An OS may not need as most of the software that comes preinstalled when an off-the-shelf image is used, which creates a bigger chance something can go wrong.

The two main type of toolchains are native and cross. Native toolchains develop code, where the host and target system have the same architecture, an example being x86. Cross-compilation toolchains will compile code on the host, targeted to run on a different architecture. Most embedded systems will use a cross toolchain for several reasons, for example embedded systems do not have the same resources and computer power as host machines (Simmonds, 2015, p. 64).

1.3 The Yocto Project

The Yocto project is an open-source project for building custom embedded Linux systems, which uses Poky as the reference build system. Poky contains both BitBake, which is used to execute commands (e.g., build software packages) and OpenEmbedded core, which contains recipes and files. Together, these components are used to build the custom Linux system, with the following parts (Gonzalez, 2015, p. 2):

- Bootloader image
- Linux kernel image
- Root filesystem image
- Toolchains and SDKs for application development

Yocto works by having layers as its building blocks. The layers are containers that hold metadata (recipes) about the specific layer. As layers are independent from one another, they can be shared and reused in different builds. The workflow of Yocto is comprised of several parts, that together make the final output. Architecturally, this can be visualised in Figure 3. These parts are summarised below, taken from the Yocto manual:

- Metadata/Inputs – The metadata and layers containing recipes are collated together and fed into the build system, where BitBake will start the generation process.
- Build System & Process Steps – The generation of each task, is done through using the BitBake commands. Each step takes in the data from the previous, resulting in the output packages, image, and the SDK.
- Output Package Feeds - The packages generated by the build system, are stored in the build directory. These packages are validated before being used in the final outputs.
- Output image & SDK – The final output images are generated. Also, an SDK recipe is created that can be integrated into other environments.

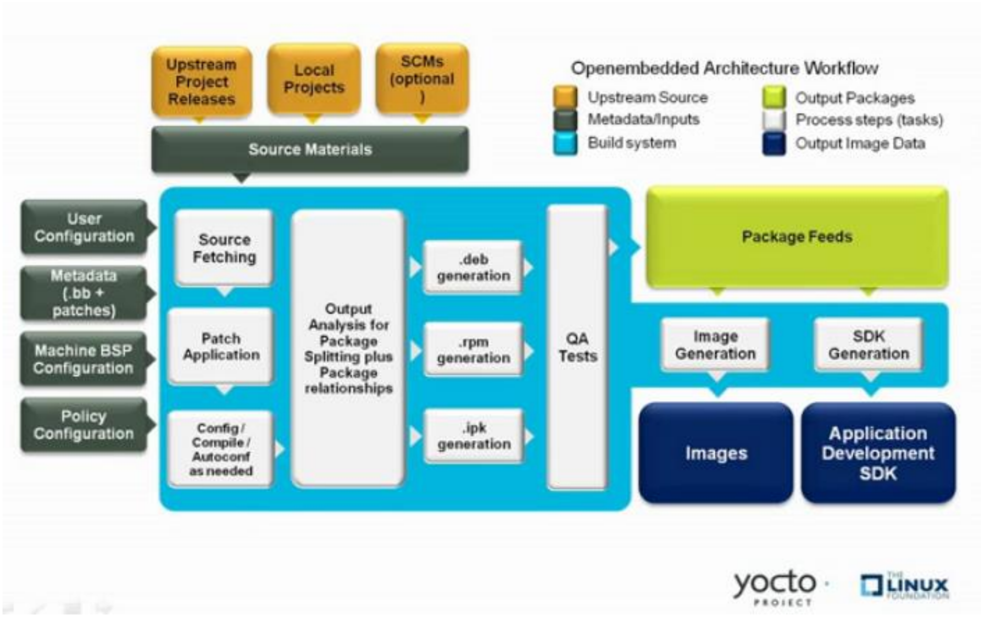
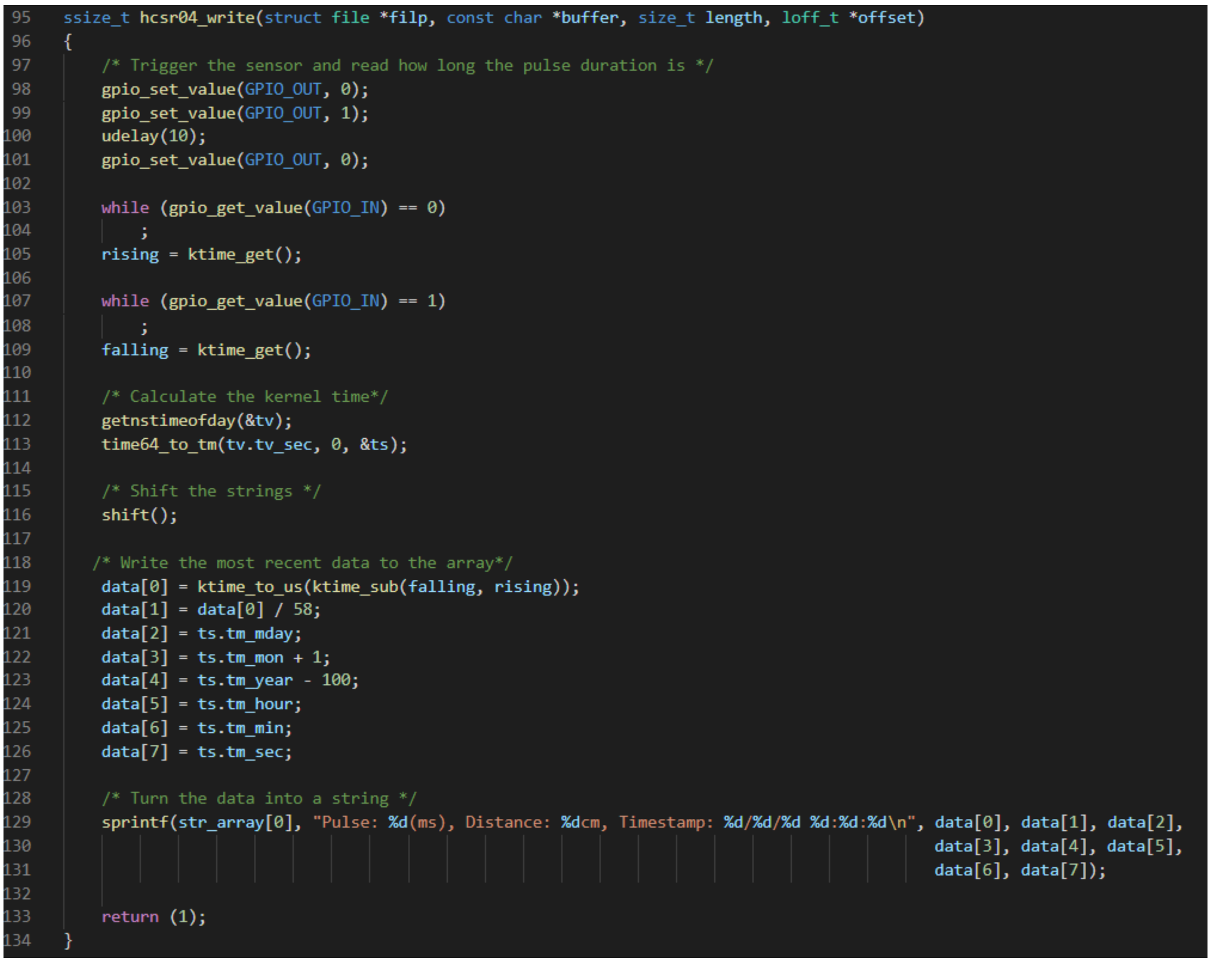


Figure 3: Yocto workflow, Scott Rifenbark (2018). Yocto Projects and Overview Manual https://www.yoctoproject.org/docs/2.5/overview-manual/overview-manual.html.

1.4 Developing a Custom Operating System with Yocto

1.4.1 Creating loadable kernel module for the HC-SR04

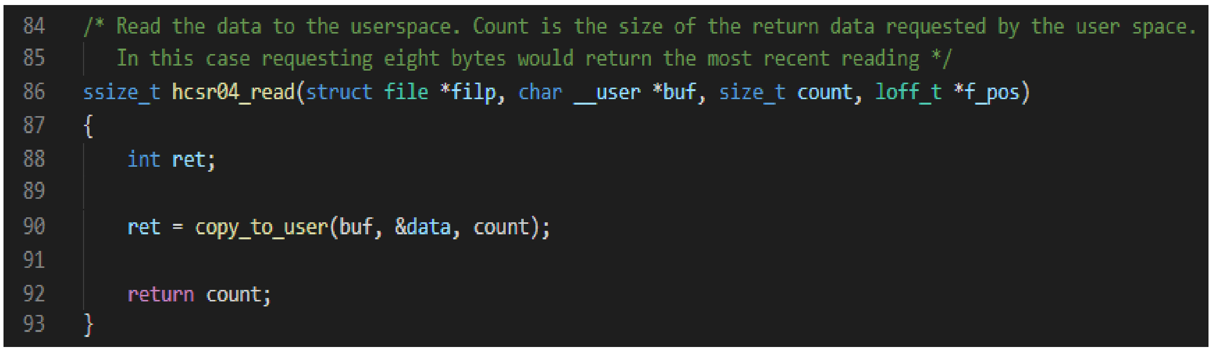
The kernel module is composed of two components. One component is the interaction with the user space done through using VFS API, and the other is interacting with the sysfs. The *write()* function (Figure 4) is used to trigger the sensor and calculate the readings. These readings are the pulse duration, distance, and timestamp. Each set of readings are stored in a string, which is used in the store function, to get the past five readings. The *kmalloc_array()* function is used to allocate memory from the kernel space to store the strings, which is done in the *hcsr04_module_init()* function (Figure 6). This is then released in *the hcsr04_module_cleanup()* function, using *kfree()*.



```c
ssize_t hcsr04_write(struct file *filp, const char *buffer, size_t length, loff_t *offset)
{
    /* Trigger the sensor and read how long the pulse duration is */
    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();

    /* Calculate the kernel time*/
    getnstimeofday(&tv);
    time64_to_tm(tv.tv_sec, 0, &ts);

    /* Shift the strings */
    shift();

   /* Write the most recent data to the array*/
    data[0] = ktime_to_us(ktime_sub(falling, rising));
    data[1] = data[0] / 58;
    data[2] = ts.tm_mday;
    data[3] = ts.tm_mon + 1;
    data[4] = ts.tm_year - 100;
    data[5] = ts.tm_hour;
    data[6] = ts.tm_min;
    data[7] = ts.tm_sec;

    /* Turn the data into a string */
    sprintf(str_array[0], "Pulse: %d(ms), Distance: %dcm, Timestamp: %d/%d/%d %d:%d:%d\n", data[0], data[1], data[2],
                                                                                            data[3], data[4], data[5],
                                                                                            data[6], data[7]);

    return (1);
}
```

Figure 4: Write() function from the kernel module.

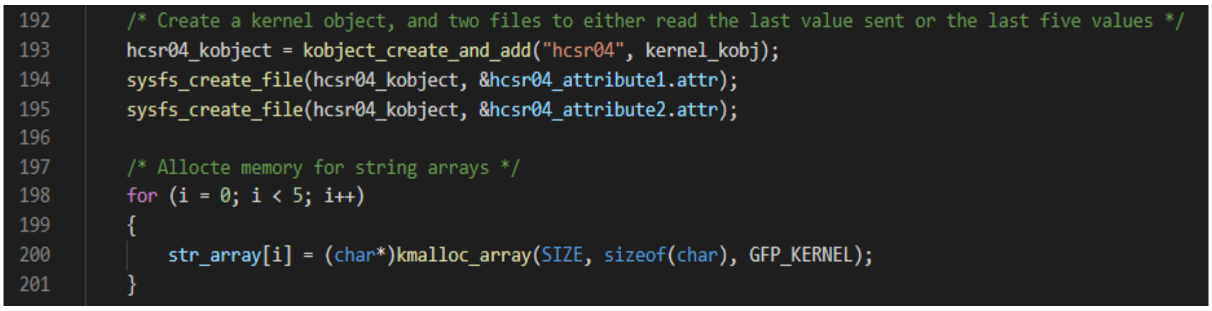
The *read()* function (Figure 5) returns the amount of data requested (in bytes) from the user space, in this case it will be 32 bytes. Instead of hard coding the number of bytes to by returned, the count argument of the function can be used, which comes from the user application.



```c
/* Read the data to the userspace. Count is the size of the return data requested by the user space.
   In this case requesting eight bytes would return the most recent reading */
ssize_t hcsr04_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
    int ret;

    ret = copy_to_user(buf, &data, count);

    return count;
}
```

Figure 5: Read() function from the kernel module.

To communicate through the sysfs a kernel object needs to be created (Figure 6). Once created, two sysfs files can be declared with their respective functions using the *sysfs_create_file()* function. The two files are created inside the kernel directory */sys/kernel/hcsr04/*, one file will show the most recent reading, and the other file will show the last five readings.

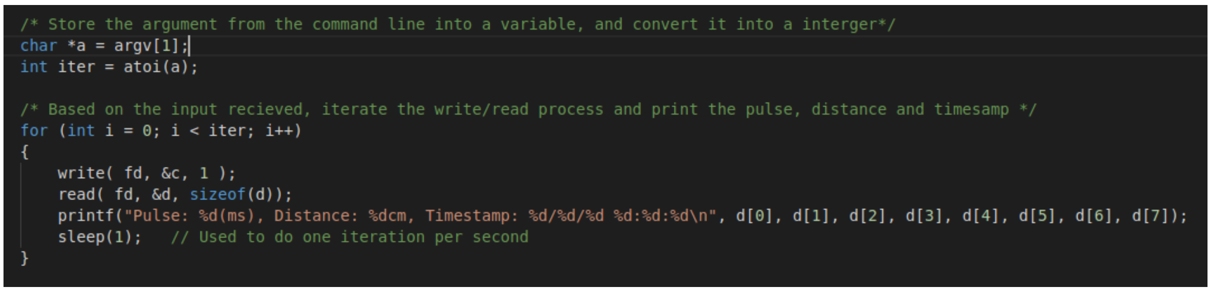


```c
    /* Create a kernel object, and two files to either read the last value sent or the last five values */
    hcsr04_kobject = kobject_create_and_add("hcsr04", kernel_kobj);
    sysfs_create_file(hcsr04_kobject, &hcsr04_attribute1.attr);
    sysfs_create_file(hcsr04_kobject, &hcsr04_attribute2.attr);

    /* Allocte memory for string arrays */
    for (i = 0; i < 5; i++)
    {
        str_array[i] = (char*)kmalloc_array(SIZE, sizeof(char), GFP_KERNEL);
    }
```

Figure 6: Kernel object created, with two files to show the last reading or the last five readings.

1.4.2 Creating a User Application

The application is simple program, that will take a command line argument for users to trigger multiple reads from the modules (Figure 7). The third argument in the *read()* function is specified in bytes and is passed to the kernel modules read function to ensure the correct number of bytes are returned.



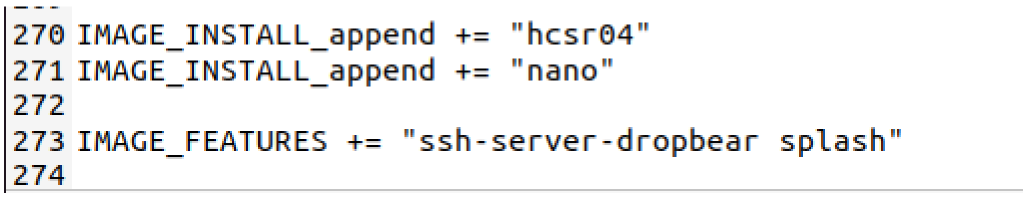
```
/* Store the argument from the command line into a variable, and convert it into a interger*/
char *a = argv[1];
int iter = atoi(a);

/* Based on the input recieved, iterate the write/read process and print the pulse, distance and timesamp */
for (int i = 0; i < iter; i++)
{
    write( fd, &c, 1 );
    read( fd, &d, sizeof(d));
    printf("Pulse: %d(ms), Distance: %dcm, Timestamp: %d/%d/%d %d:%d:%d\n", d[0], d[1], d[2], d[3], d[4], d[5], d[6], d[7]);
    sleep(1);   // Used to do one iteration per second
}
```

Figure 7: User application developed for the HC-SR04 kernel module.

1.4.3 Adding Additional Features

Additional features can be added onto the build of the system, to further customise OS. These features are adding a Dropbear SSH server, adding a splash screen to show when booting up, and loading nano text editor.

The nano recipe can either be downloaded from the OpenEmbedded Index page. Using IMAGE_INSTALL_append in the local.conf file will load this recipe during the bootup. The Dropbear SSH server and splash screen can also be added in a slightly different way (Figure 8). With the splash screen the image seen can be changed by running a script to convert the given image into the correct format (*Replace Default Splash Screen in Yocto*, 2014).



```
IMAGE_INSTALL_append += "hcsr04"
IMAGE_INSTALL_append += "nano"

IMAGE_FEATURES += "ssh-server-dropbear splash"

```

Figure 8: Local.conf file, showing the additional features being added.

1.5 Testing the Custom Operating System

1.5.1 Testing Additional Features

The splash screen is shown when the system is first powered on, and because of this it can be easily missed due to how fast the system takes to boot up. Rebooting the board from the command line, will show the splash screen on the next powerup Figure 9.

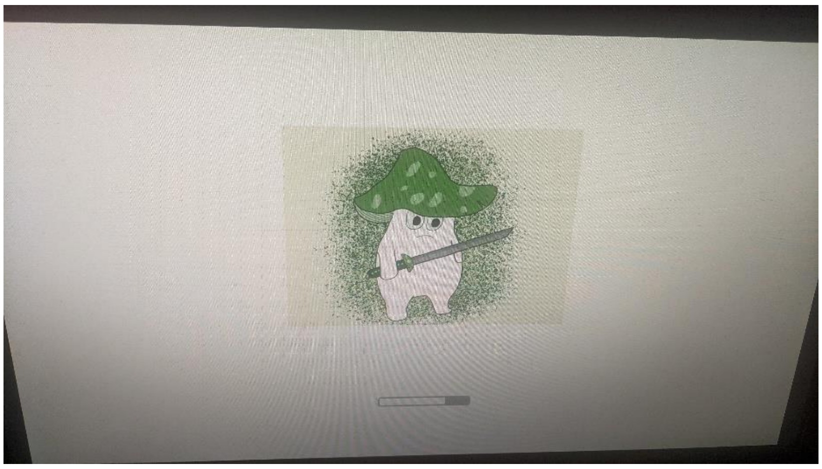
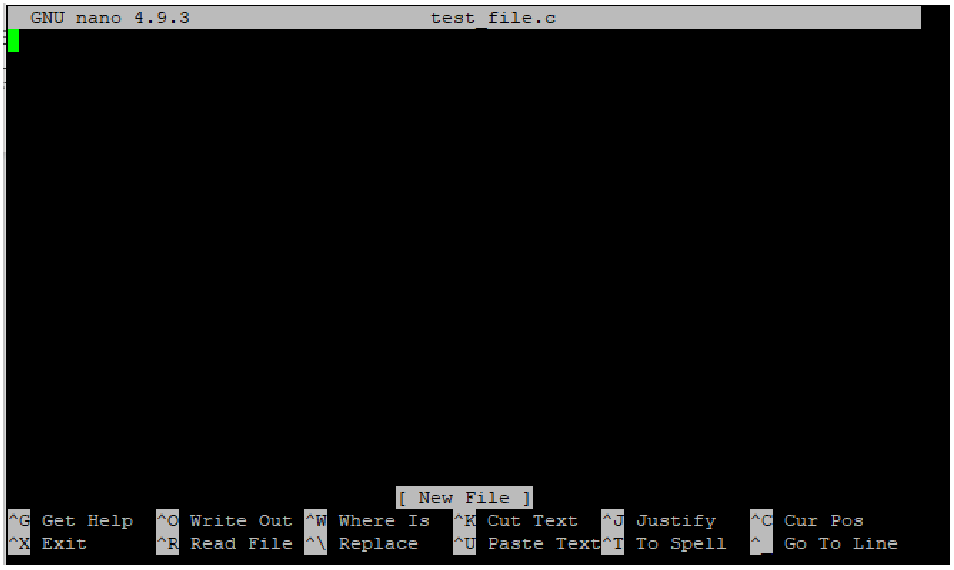


Figure 9: Picture of the splash screen.

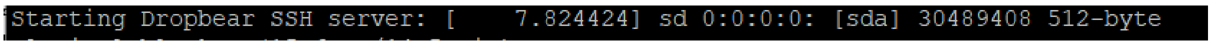
To check if the nano text editor is loaded, this can simply be done by writing nano in the terminal, followed by the name of an existing or new file to edit (Figure 10).



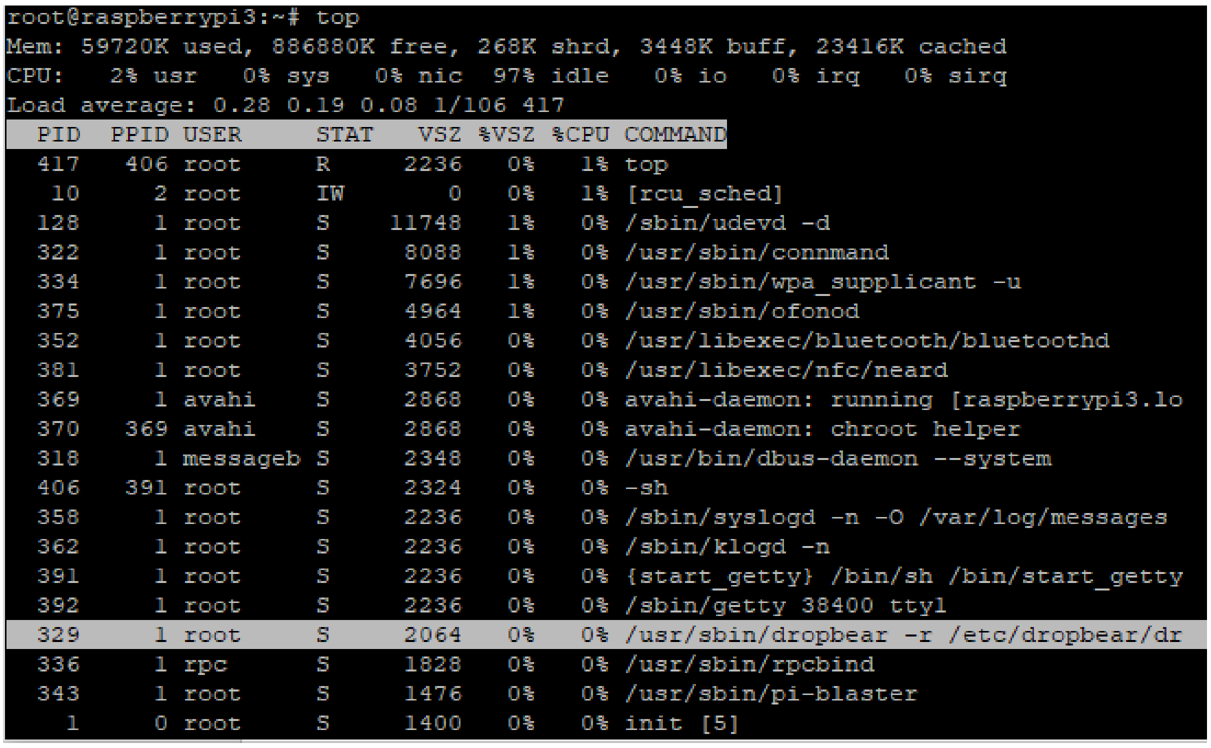
```
  GNU nano 4.9.3                    test_file.c





                                  [ New File ]
^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos
^X Exit       ^R Read File  ^\ Replace    ^U Paste Text^T To Spell    ^_ Go To Line
```

Figure 10: Using nano to open a file.

Lastly the Dropbear SSH sever can be seen being loaded during the bootup sequence. Using Linux commands can also be used to check if the Dropbear server is running. For example, using the "top" command, will show if the Dropbear SSH server is being managed by the kernel (Figure 11).



```
Starting Dropbear SSH server: [    7.824424] sd 0:0:0:0: [sda] 30489408 512-byte
```

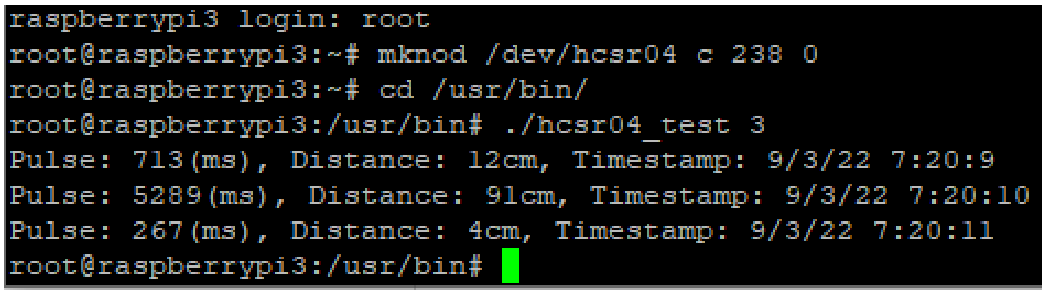


```
root@raspberrypi3:~# top
Mem: 59720K used, 886880K free, 268K shrd, 3448K buff, 23416K cached
CPU:   2% usr   0% sys   0% nic  97% idle   0% io   0% irq   0% sirq
Load average: 0.28 0.19 0.08 1/106 417
  PID  PPID USER     STAT   VSZ %VSZ %CPU COMMAND
  417   406 root     R     2236   0%   1% top
   10     2 root     IW       0   0%   1% [rcu_sched]
  128     1 root     S    11748   1%   0% /sbin/udevd -d
  322     1 root     S     8088   1%   0% /usr/sbin/connmand
  334     1 root     S     7696   1%   0% /usr/sbin/wpa_supplicant -u
  375     1 root     S     4964   1%   0% /usr/sbin/ofonod
  352     1 root     S     4056   0%   0% /usr/libexec/bluetooth/bluetoothd
  381     1 root     S     3752   0%   0% /usr/libexec/nfc/neard
  369     1 avahi    S     2868   0%   0% avahi-daemon: running [raspberrypi3.lo
  370   369 avahi    S     2868   0%   0% avahi-daemon: chroot helper
  318     1 messageb S     2348   0%   0% /usr/bin/dbus-daemon --system
  406   391 root     S     2324   0%   0% -sh
  358     1 root     S     2236   0%   0% /sbin/syslogd -n -O /var/log/messages
  362     1 root     S     2236   0%   0% /sbin/klogd -n
  391     1 root     S     2236   0%   0% {start_getty} /bin/sh /bin/start_getty
  392     1 root     S     2236   0%   0% /sbin/getty 38400 tty1
  329     1 root     S     2064   0%   0% /usr/sbin/dropbear -r /etc/dropbear/dr
  336     1 rpc      S     1828   0%   0% /usr/sbin/rpcbind
  343     1 root     S     1476   0%   0% /usr/sbin/pi-blaster
    1     0 root     S     1400   0%   0% init [5]
```

Figure 11: Boot up showing the Dropbear SSH server starting (top) and running the "top" command (bottom).

1.5.2 Testing the User Application

The user application has automatically been loaded into the /usr/bin directory. This file takes an argument, that is used to trigger the HC-SR04 sensor multiple times. For example, Figure 12 shows the output of running the application three times.

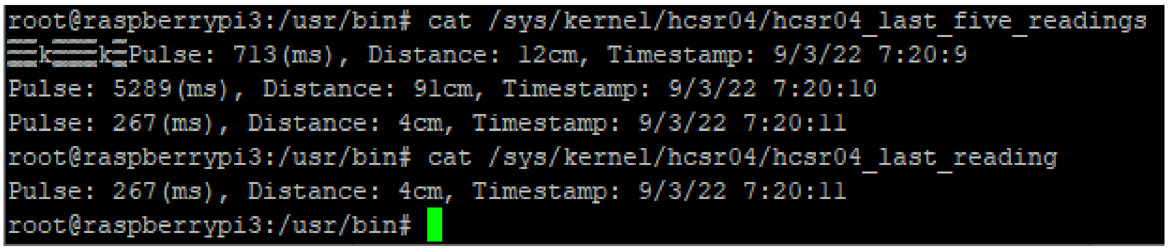


```
raspberrypi3 login: root
root@raspberrypi3:~# mknod /dev/hcsr04 c 238 0
root@raspberrypi3:~# cd /usr/bin/
root@raspberrypi3:/usr/bin# ./hcsr04_test 3
Pulse: 713(ms), Distance: 12cm, Timestamp: 9/3/22 7:20:9
Pulse: 5289(ms), Distance: 91cm, Timestamp: 9/3/22 7:20:10
Pulse: 267(ms), Distance: 4cm, Timestamp: 9/3/22 7:20:11
root@raspberrypi3:/usr/bin#
```

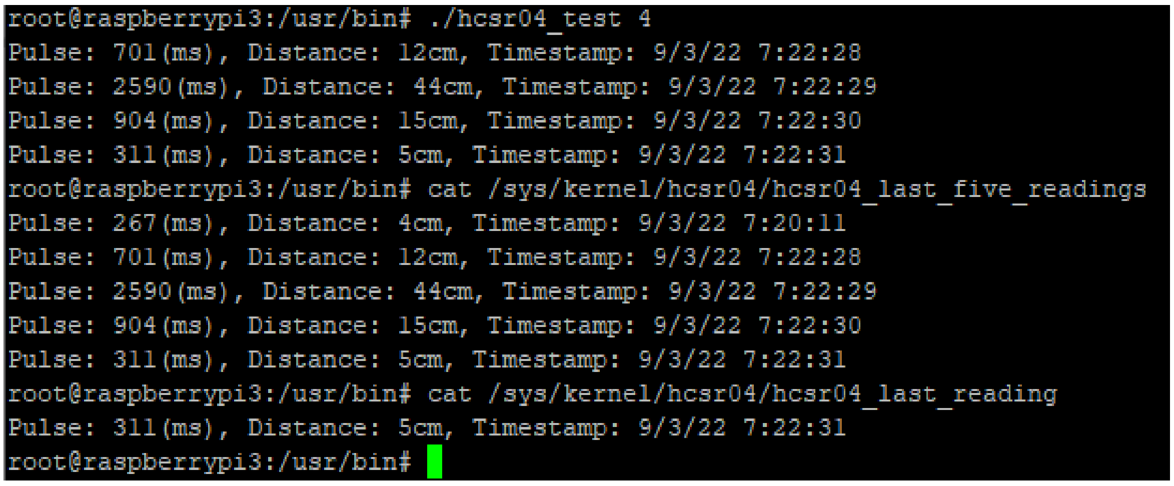
Figure 12: Running the user application three times.

1.5.3 Testing the SYSFS

After running the user application using the sysfs can show the same output, produced from the kernel module. In the /sys/kernel/hcsr04/ directory there is two file that be run. One file shows the last reading produced by the sensor, and the other file shows the last five readings. For example, after running the user application three times, the output can be displayed using the hcsr04_last_five_readings file that displays the last five readings. If the user application is triggered again, to read the sensor another four times, the oldest reading from the previous run would be the at the top (Figure 13).



```
root@raspberrypi3:/usr/bin# cat /sys/kernel/hcsr04/hcsr04_last_five_readings
≡k≡≡k≡Pulse: 713(ms), Distance: 12cm, Timestamp: 9/3/22 7:20:9
Pulse: 5289(ms), Distance: 91cm, Timestamp: 9/3/22 7:20:10
Pulse: 267(ms), Distance: 4cm, Timestamp: 9/3/22 7:20:11
root@raspberrypi3:/usr/bin# cat /sys/kernel/hcsr04/hcsr04_last_reading
Pulse: 267(ms), Distance: 4cm, Timestamp: 9/3/22 7:20:11
root@raspberrypi3:/usr/bin#
```



```
root@raspberrypi3:/usr/bin# ./hcsr04_test 4
Pulse: 701(ms), Distance: 12cm, Timestamp: 9/3/22 7:22:28
Pulse: 2590(ms), Distance: 44cm, Timestamp: 9/3/22 7:22:29
Pulse: 904(ms), Distance: 15cm, Timestamp: 9/3/22 7:22:30
Pulse: 311(ms), Distance: 5cm, Timestamp: 9/3/22 7:22:31
root@raspberrypi3:/usr/bin# cat /sys/kernel/hcsr04/hcsr04_last_five_readings
Pulse: 267(ms), Distance: 4cm, Timestamp: 9/3/22 7:20:11
Pulse: 701(ms), Distance: 12cm, Timestamp: 9/3/22 7:22:28
Pulse: 2590(ms), Distance: 44cm, Timestamp: 9/3/22 7:22:29
Pulse: 904(ms), Distance: 15cm, Timestamp: 9/3/22 7:22:30
Pulse: 311(ms), Distance: 5cm, Timestamp: 9/3/22 7:22:31
root@raspberrypi3:/usr/bin# cat /sys/kernel/hcsr04/hcsr04_last_reading
Pulse: 311(ms), Distance: 5cm, Timestamp: 9/3/22 7:22:31
root@raspberrypi3:/usr/bin#
```

Figure 13: Reading the last five readings from hcsr04_last_five_readings file, and the last reading from the hcsr04_1 file (top). Running the user application and reading from the kernel files again (bottom).

1.6 Discussion and Summary

The overall process of creating a custom embedded Linux OS using Yocto has straightforward. Most errors that were present during the development, was mainly to do with incorrect syntax and packages not being installed. During the development of the kernel module, any errors was primarily to do with the code, and using the Linux libraries incorrectly. Learning debugging methods for creating kernel drivers, as well as using Yocto could have sped up the overall process.

The performance of the system works as designed. However, when running the application for the first time, the terminal sometimes shows some garbage text at the start (Figure 13). Lastly, testing the image on a host machine with an ethernet port would have been a better way to test that the SSH is working.

1.7 References

Gonzalez, A. (2015). Embedded Linux projects using yocto project cookbook. Packt Publishing.

Hallinan, C. (2010). Embedded Linux primer: A practical real-world approach (2nd ed.). Prentice Hall.

Simmonds, C. (2015). Mastering embedded Linux programming. Packt Publishing.

Replace default splash screen in Yocto. (2014, March 14). Easy Linux. Retrieved February 16, 2022, from https://easylinuxji.blogspot.com/2019/03/replace-default-splash-screen-in-yocto.html