# Handling HC-SR04 ranging sensor using linux kernel modules

## goal

The goal of this Lab is to build a loadable kernel module using the Yocto build environment that provides access to the HC-SR04 ranging sensor.

## prerequisites

To follow this Lab, you need:

1. Raspberry Pi 3 board full;
2. One HC-SR04 ultrasonic ranging sensor;
3. Micro USB cable;
4. 8 GB Micro SD card;
5. A PC provided with Ubuntu Desktop 14.04 LTS, or a virtual machine hosting Ubuntu Desktop 14.04 LTS;
6. A Micro SD card reader attached to the PC/virtual machine;
7. (**Optional**) Micro HDMI cable.

## Introduction

The Raspberry Pi 3 offers a number of I/O accessible through the connectors labeled J8 in the following picture:

|  |
| --- |
|  |

In this Lab, a loadable kernel module is implemented that program GPIO21 as output to be connected to the HC-SR04 Tring input and GPIO20 as input to be connected to the HC-SR04 Echo output.

Furthermore, the HC-SR04 Vcc input shall be connected to the 5V pin on J8, and the HC-SR04 GND shall be connected to any GND pin (e.g., the one on J8).

## workplace setup

Assuming you completed the previous lab, move to the directory ***raspberrypi3*** and prepare the build environment:

cd ~/raspberrypi3/

source sources/poky/oe-init-build-env rpi-build

You now have the system ready for building the embedded Linux distribution for the Raspberry Pi 3 board.

## Preparing the recipes

In this Lab, we are adding a new device driver to be compiled as a loadable kernel module, as you will have to do in case you are asked to customize the board support package for the Raspberry Pi 3 to support a new I/O device attached to the system.

For this purpose, we will add a new Yocto recipe to the ***meta-raspberrypi*** layer. For this purpose, create the directory as follows:

cd ~/raspberrypi3/sources/meta-raspberrypi/recipes-kernel

mkdir -p hcsr04-mod/files

In ***hcsr04-mod***, put a file named ***hcsr04\_1.0.bb*** with the following content:

|  |  |
| --- | --- |
|  0: 1: 2: 3: 4: 5: 6: 7: 8: 9: | DESCRIPTION = "hcsr04 driver"LICENSE = "GPLv2+"LIC\_FILES\_CHKSUM = "file://${BPN}.c;md5=94af90acd77ed5583caa62cbe903e559"inherit modulePR = "r0"SRC\_URI = "file://Makefile file://${BPN}.c"S = "${WORKDIR}" |

This file describes the property of the recipe, and in particular:

* Line 0 gives a simple description of the recipe purpose;
* Lines 1-2 give the reference to the license under which the recipe is distributed and provide the indication of the license file and its checksum;
* Line 4 indicates that the recipe inherits the properties of the recipes to build kernel modules;
* Line 6 tells for which machine layer the recipe is intended;
* Lines 8 lists the source files needed to build the recipe;
* Line 9 sets the symbol pointing to the working directory used to perform the build operations.

In ***hcsr04-mod/files***, create a file named ***hcsr04.c*** with the following content:

|  |  |
| --- | --- |
|  | #include <linux/module.h>#include <linux/init.h>#include <linux/kernel.h>#include <linux/types.h>#include <linux/kdev\_t.h>#include <linux/fs.h>#include <linux/cdev.h>#include <asm/uaccess.h>#include <linux/gpio.h>#include <linux/interrupt.h>#include <linux/kobject.h>#include <linux/sysfs.h>#include <linux/time.h>#include <linux/ktime.h>#include <asm/delay.h>#include <linux/delay.h>#define GPIO\_OUT 20 // GPIO20#define GPIO\_IN 21 // GPIO21static dev\_t hcsr04\_dev;struct cdev hcsr04\_cdev;static int hcsr04\_lock = 0;static struct kobject \*hcsr04\_kobject;static ktime\_t rising, falling;int hcsr04\_open(struct inode \*inode, struct file \*file){ int ret = 0; printk( KERN\_INFO "hcsr04\_dev: %s\n", \_\_func\_\_ ); if( hcsr04\_lock > 0 ) { ret = -EBUSY; } else hcsr04\_lock++; return( ret );}int hcsr04\_close(struct inode \*inode, struct file \*file){ printk( KERN\_INFO "hcsr04\_dev: %s\n", \_\_func\_\_ ); hcsr04\_lock = 0; return( 0 );}ssize\_t hcsr04\_read(struct file \*filp, char \_\_user \*buf, size\_t count, loff\_t \*f\_pos){ int ret; int pulse; printk( KERN\_INFO "hcsr04\_dev: %s\n", \_\_func\_\_ ); pulse = (int)ktime\_to\_us( ktime\_sub( falling, rising ) ); ret = copy\_to\_user( buf, &pulse, 4 ); return 4;}ssize\_t hcsr04\_write(struct file \*filp, const char \*buffer, size\_t length, loff\_t \* offset){ printk( KERN\_INFO "hcsr04\_dev: %s\n", \_\_func\_\_ ); 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(); return( 1 );}struct file\_operations hcsr04\_fops = { .owner = THIS\_MODULE, .read = hcsr04\_read, .write = hcsr04\_write, .open = hcsr04\_open, .release = hcsr04\_close,};static ssize\_t hcsr04\_show(struct kobject \*kobj, struct kobj\_attribute \*attr, char \*buf){ printk( KERN\_INFO "hcsr04\_dev: %s\n", \_\_func\_\_ ); return sprintf(buf, "%d\n", ktime\_to\_us(ktime\_sub(falling,rising)));}static ssize\_t hcsr04\_store(struct kobject \*kobj, struct kobj\_attribute \*attr, char \*buf, size\_t count){ printk( KERN\_INFO "hcsr04\_dev: %s\n", \_\_func\_\_ ); return 1;}static struct kobj\_attribute hcsr04\_attribute =\_\_ATTR(hcsr04, 0660, hcsr04\_show, hcsr04\_store);static int \_\_init hcsr04\_module\_init(void){ char buffer[64]; int ret = 0; printk(KERN\_INFO "Loading hcsr04\_module\n"); alloc\_chrdev\_region(&hcsr04\_dev, 0, 1, "hcsr04\_dev"); printk(KERN\_INFO "%s\n", format\_dev\_t(buffer, hcsr04\_dev)); cdev\_init(&hcsr04\_cdev, &hcsr04\_fops); hcsr04\_cdev.owner = THIS\_MODULE; cdev\_add(&hcsr04\_cdev, hcsr04\_dev, 1); if( gpio\_request( GPIO\_OUT, "hcsr04\_dev" ) ) { printk( KERN\_INFO "hcsr04\_dev: %s unable to get GPIO\_OUT\n", \_\_func\_\_ ); ret = -EBUSY; goto Done; } if( gpio\_request( GPIO\_IN, "hcsr04\_dev" ) ) { printk( KERN\_INFO "hcsr04\_dev: %s unable to get GPIO\_IN\n", \_\_func\_\_ ); ret = -EBUSY; goto Done; } if( gpio\_direction\_output( GPIO\_OUT, 0 ) < 0 ) { printk( KERN\_INFO "hcsr04\_dev: %s unable to set GPIO\_OUT as output\n", \_\_func\_\_ ); ret = -EBUSY; goto Done; } if( gpio\_direction\_input( GPIO\_IN ) < 0 ) { printk( KERN\_INFO "hcsr04\_dev: %s unable to set GPIO\_IN as input\n", \_\_func\_\_ ); ret = -EBUSY; goto Done; } hcsr04\_kobject = kobject\_create\_and\_add("hcsr04", kernel\_kobj); if(!hcsr04\_kobject) { ret = -ENOMEM; goto Done; } ret = sysfs\_create\_file(hcsr04\_kobject, &hcsr04\_attribute.attr); if( ret ) { printk( KERN\_INFO "failed to create the foo file in /sys/kernel/hcsr04\n"); ret = -ENOMEM; goto Done; }Done: return ret;}static void \_\_exit hcsr04\_module\_cleanup(void){ printk(KERN\_INFO "Cleaning-up hcsr04\_dev.\n"); gpio\_free( GPIO\_OUT ); gpio\_free( GPIO\_IN ); hcsr04\_lock = 0; cdev\_del(&hcsr04\_cdev); unregister\_chrdev\_region( hcsr04\_dev, 1 ); kobject\_put( hcsr04\_kobject );}module\_init(hcsr04\_module\_init);module\_exit(hcsr04\_module\_cleanup);MODULE\_AUTHOR("Your name");MODULE\_LICENSE("GPL");  |

Finally, in the same directory, place a file name ***Makefile*** with the following content:

|  |  |
| --- | --- |
| 0:1:2:3:4:5:6:7:8:9:10:11:12:13:14: | obj-m := hcsr04.oSRC := $(shell pwd)all: $(MAKE) -C $(KERNEL\_SRC) M=$(SRC)modules\_install: $(MAKE) INSTALL\_MOD\_DIR=kernel/drivers/my\_mod -C $(KERNEL\_SRC) M=$(SRC) modules\_installclean: rm -f \*.o \*~ core .depend .\*.cmd \*.ko \*.mod.c rm -f Module.markers Module.symvers modules.order rm -rf .tmp\_versions Modules.symvers |

The file describes how the source code shall be built and where the output shall be placed, in particular:

* Lines 0-5 define which is the object module to be created, locate the current directory, and invoke the build command, which refers to the source and kernel source symbols Yocto maintains;
* Lines 7-8 define which operation to perform for installing the loadable kernel module resulting from the compilation. In this example, the object will be installed in the root file system in the /lib/modules/<kernel version>/kernel/drivers/my\_mod directory.
* Lines 10-14 define the operation to perform when cleaning the build outputs (i.e., removing the temporary files and the compilation outputs).

Once these operations are completed, you have to tell the machine layer configuration that the new driver is needed. For this purpose, edit the file

 cd ~/raspberrypi3/rpi-build/conf/local.conf

Add the following statement as a last line of the file:

IMAGE\_INSTALL\_append += "hcsr04"

This line tells Yocto that when building Linux, the newly created device driver shall be built and that it shall be included in the root file system.

## Building and deploying the new system

You are now ready to build the new system as follows:

cd ~/raspberrypi3/rpi-build

bitbake -c clean rpi-basic-image

bitbake rpi-basic-image

During the build, the compiler will recognize a discrepancy in the license reference. Copy and paste the suggested reference (“the new md5 checksum is..”) into your hcsr04\_1.0.bb file and start the build again.

After a while, a new Micro SD card image would be available, which you can deploy in the Micro SD as follows (assuming the Micro SD is available to the PC as /dev/sdN). Alternatively, use a program of your preference to flash the image.

First, run the:

sudo fdisk -l

command to determine which device to flash to (plug in and unplug the SD card to determine which device it is). For this example, the SD card is under the name “sdc” (this may be different in your environment). Next, ensure that the device is unmounted. This can be done using the command:

sudo umount /dev/sdc\*

Once this is done, the following command can be used to copy the image across to the SD card (substitute any folder names and device names to ensure they are relevant to your specific environment).

sudo dd bs=1M if=/home/user/raspberryPi3/rpi-build/tmp-glibc/deploy/images/raspberrypi3/rpi-basic-image-raspberrypi3.rpi-sdimg of=/dev/sdc

Note that if not done properly, the image being flashed across to the SD card may cause problems when attempting to turn on the board. If this is the case, it may be worth retrying the process again and ensuring that it is done properly or use a flash program to automate the process.

While the SD card is still connected to the development host, use the following lines to navigate to the etc folder on the SD card (assuming the device has now been mounted).

cd media/user/SD\_name/etc – Use the ls command to find the name of the SD card in the user folder.

Then, use a terminal text editor to open the shadow file.

sudo vi shadow

Or

sudo gedit shadow

Check that there are no characters between the first two colons in the first line. If there is, remove it so that the first line looks like this:

Root::17728:0:99999:7:::

Exit the text editor by entering :x or simply closing the application!

## Running the module

After booting the new Linux system, you can check whether the build process was completed successfully. After logging into the Raspberry Pi 3, you can type the following commands:

root@raspberrypi3:/# ls -la /lib/modules/4.1.21/kernel/drivers/my\_mod

drwxr-xr-x 2 root root 1024 May 15 08:47 .

drwxr-xr-x 43 root root 1024 May 10 12:09 ..

-rw-r--r-- 1 root root 9080 May 12 03:02 gpio.ko

-rw-r--r-- 1 root root 10140 May 15 08:22 hcsr04.ko

-rw-r--r-- 1 root root 5384 May 11 08:05 hello.ko

root@raspberrypi3:/#

The directory in the root file system contains ***hcsr04.ko***, which is the kernel object containing the binary code for the loadable kernel module.

You can now insert the module in the kernel as follows:

root@raspberrypi3:/# insmod /lib/modules/4.1.21/kernel/drivers/my\_mod/hcsr04.ko

[ 143.363069] Loading hcsr04\_module

[ 143.366466] 243:0

The module is loaded with major number 243 and minor number 0. If you are not connected to the serial console, you can see the messages logged by the module using the dmesg command.

To test the module, you have to first create the associated device file:

root@raspberrypi3:/# mknod /dev/hcsr04 c 243 0

You can now communicate with the module through the Linux command line.

As an example, you can issue the following command, observing the following output:

root@raspberrypi3:/# echo 1 > /dev/hcsr04

[ 223.733423] hcsr04\_dev: hcsr04\_open

[ 223.737076] hcsr04\_dev: hcsr04\_write

[ 223.753109] hcsr04\_dev: hcsr04\_write

[ 223.769147] hcsr04\_dev: hcsr04\_close

If you have connected the Raspberry Pi 3 to the HC-SR04 ranging sensor, you can read the measured echo pulse duration as follows:

root@raspberrypi3:~# cat /sys/kernel/hcsr04/hcsr04

[ 279.097877] hcsr04\_dev: hcsr04\_show

11994

The value 11.994 s corresponding to about 206 cm has been measured with the HC-SR04 sensor resting on a table pointing toward the ceiling in a room 300 cm tall.

In this example, we are using the virtual file system API to trigger the sensor and sysfs to read the measured distance.

You can put an object at different distance with respect to the sensor and validate the correct operations of the module by repeating the trigger operation.

To test the module, we can also write a simple test application called ***test\_hcsr04.c*** with the following code:

#include <stdio.h>

#include <string.h>

#include <errno.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

int main(int argc, char \*\*argv)

{

 char \*app\_name = argv[0];

 char \*dev\_name = "/dev/hcsr04";

 int fd = -1;

 char c;

 int d;

 if( (fd = open(dev\_name, O\_RDWR)) < 0 )

 {

 fprintf(stderr, "%s: unable to open %s: %s\n", app\_name, dev\_name, strerror(errno));

 return( 1 );

 }

 c = 1;

 write( fd, &c, 1 );

 read( fd, &d, 4 );

 printf( "%d: %f\n", d, d/58.0 );

 close( fd );

 return 0;

}

The code shall be built on the host machine, issuing the following command:

arm-linux-gnueabihf-gcc -o test\_hcsr04 test\_hcsr04.c

The obtained program, ***test\_hcsr04***, shall be executed on the Raspberry Pi 3 target, for example, by copying it from the development host onto a USB stick that will be then plugged into the Raspberry Pi 3 USB port.

The test program can either be copied across to the board via the SD card, or it can be executed as follows, after mounting first the USB stick to /media/:

root@raspberrypi3:~# /media/test\_hcsr04

13552: 233.655172

The test application interacts with the driver through the virtual file system: it triggers the sensor and then it reads the measured distance. As we are in the same condition as before, we obtain the same result.

For removing the module, you can act as follows:

root@raspberrypi3:/# rmmod hcsr04

After removing the module, you can verify that the hcsr04 directory is no longer available in /sys/kernel.

# post-lab practice

Now you can play your game on your Raspberry Pi 3 board with buttons as controls. To make things more interesting, we are going to utilize the HC-SR04 ranging sensor to control the game. This is to further enhance your practical skills on kernel programming. Also, you may well need to write a program in user space to achieve the ultimate goal of this challenge session.

**Q: Amend the test program given in the Lab to send key events according to the output from the HC-SR04 ranging sensor. For instance, moving your hand closer the sensor will invoke LEFT key event and moving further will cause RIGHT key event. You will also need to amend the kernel module for a complete and smooth experience. (Hint: consider the use of uinput)**

A:. The program measures and calculates the distance from the sensor to its obstacle; hence, we can easily modify it and send key strokes according to the measured distance. The most important part is how we can generate key events in user space. It appears that uinput is the answer. uinput stands for “user input,” which is a virtual input device that you can find at /dev directory. To enable uinput, simply include <linux/uinput.h> interface in your program.

The example solution is provided in C++ but not too different from the standard C. Here is the completed code:

|  |  |  |
| --- | --- | --- |
| 0:1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:20:21:22:23:24:25:26:27:28:29:30:31:32:33:34:35:36:37:38:39:40:41:42:43:44:45:46:47:48:49:50:51:52:53:54:55:56:57:58:59:60:61:62:63:64:65:66:67:68:69:70:71:72:73:74:75:76:77:78:79:80:81:82:83:84:85:86:87:88:89:90:91:92:93:94:95:96:97:98:99:100:101:102:103:104:105:106:107:108:109:110:111:112:113:114:115:116:117:118:119:120:121:122:123:124:125:126:127:128:129:130:131:132:133:134:135:136:137:138:139:140:141:142:143:144:145:146:147:148:149:150:151:152:153:154:155:156:157:158:159:160:161:162:163:164:165:166:167:168:169:170:171:172:173:174:175: | #include <cstdlib>#include <iostream>#include <string>#include <stdio.h>#include <stdlib.h>#include <string.h>#include <unistd.h>#include <fcntl.h>#include <errno.h>#include <time.h>#include <linux/input.h>#include <linux/uinput.h>using namespace std;int SetupUinputDevice(){ // file descriptor for the uinput device int fd\_uinput\_device; fd\_uinput\_device = open("/dev/uinput", O\_WRONLY | O\_NONBLOCK | O\_NDELAY); if(fd\_uinput\_device < 0) { std::cout << "Error : open file descriptor /dev/uinput : " << strerror(errno) << std::endl;  return -1; } // create and initialise uinput\_user\_dev struct  struct uinput\_user\_dev dev\_key; memset(&dev\_key, 0, sizeof(uinput\_user\_dev)); snprintf(dev\_key.name, UINPUT\_MAX\_NAME\_SIZE, "ranging\_sensor"); dev\_key.id.bustype = BUS\_USB; dev\_key.id.vendor = 0x01; dev\_key.id.product = 0x02; dev\_key.id.version = 1; // configure/set key press and release events if(ioctl(fd\_uinput\_device, UI\_SET\_EVBIT, EV\_KEY) < 0) { std::cout << "Error : ioctl : UI\_SET\_EVBIT for EV\_KEY " << strerror(errno) << std::endl; return -1; } // enable set of key board events for(int iEvent=0; iEvent < 254; iEvent++) { if(ioctl(fd\_uinput\_device, UI\_SET\_KEYBIT, iEvent) < 0) { std::cout << "Error : ioctl : UI\_SET\_KEYBIT for event ID: " << iEvent << " : " << strerror(errno) << std::endl; } } // enable synchronization events if(ioctl(fd\_uinput\_device, UI\_SET\_EVBIT, EV\_SYN) < 0) { std::cout << "Error : ioctl : UI\_SET\_EVBIT for EV\_SYN: " << strerror(errno) << std::endl; return -1; } // write the uinput\_user\_dev structure into the device file descriptor if(write(fd\_uinput\_device, &dev\_key, sizeof(uinput\_user\_dev)) < 0) { std::cout << "Error : failed to write uinput\_user\_dev structure into the device file descriptor: " << strerror(errno) << std::endl; return -1; } // Create the end point file descriptor for user input device descriptor. if(ioctl(fd\_uinput\_device, UI\_DEV\_CREATE) < 0) { std::cout << "Error : failed to create end point for user input device: " << strerror(errno) << std::endl; return -1; } return fd\_uinput\_device;}int CloseUinputDevice(int fd\_dev){ if(ioctl(fd\_dev, UI\_DEV\_DESTROY) < 0) { std::cout << "Error : ioctl failed: UI\_DEV\_DESTROY : " << strerror(errno) << std::endl; return -1; } if(close(fd\_dev) < 0) { std::cout << "Error : close device file descriptor : " << strerror(errno) << std::endl; return -1; }}// function that sends key eventsint emit(int fd\_dev, int event\_type, int key\_code, int value){ // input\_event struct member for input events struct input\_event key\_input\_event; memset(&key\_input\_event, 0, sizeof(input\_event)); // set event values key\_input\_event.type = event\_type; key\_input\_event.code = key\_code; key\_input\_event.value = value; if(write(fd\_dev, &key\_input\_event, sizeof(input\_event)) < 0) { std::cout << "Error writing input events to the device descriptor: " << strerror(errno) << std::endl; return -1; } return 0;}int main(int argc, char\*\* argv) { int fd\_uinput = SetupUinputDevice(); sleep(2);  char \*app\_name = argv[0];  const char \*dev\_name = "/dev/hcsr04";  int fd = -1;  char c = 1;  int time; int distance;  if(fd\_uinput < 0) { std::cout << "Error in setup file descriptor for uinput device..." << std::endl; return -1; }  if( (fd = open(dev\_name, O\_RDWR)) < 0 )  { fprintf(stderr, "%s: unable to open %s: %s\n", app\_name, dev\_name, strerror(errno));  return -1;  } while (1) {  write( fd, &c, 1 ); usleep (10000); read( fd, &time, 4 ); distance = time/58.0; // effective distance is within 50 and 15 is the boundary if (distance>15 && distance <50) { emit(fd\_uinput, EV\_KEY, KEY\_RIGHT, 0); emit(fd\_uinput, EV\_SYN, SYN\_REPORT, 0); emit(fd\_uinput, EV\_KEY, KEY\_LEFT, 1); emit(fd\_uinput, EV\_SYN, SYN\_REPORT, 0); }  else if (distance <= 15) { emit(fd\_uinput, EV\_KEY, KEY\_LEFT, 0); emit(fd\_uinput, EV\_SYN, SYN\_REPORT, 0); emit(fd\_uinput, EV\_KEY, KEY\_RIGHT, 1); emit(fd\_uinput, EV\_SYN, SYN\_REPORT, 0); } else  { emit(fd\_uinput, EV\_KEY, KEY\_RIGHT, 0); emit(fd\_uinput, EV\_SYN, SYN\_REPORT, 0); emit(fd\_uinput, EV\_KEY, KEY\_LEFT, 0); emit(fd\_uinput, EV\_SYN, SYN\_REPORT, 0); }  } close( fd ); CloseUinputDevice(fd\_uinput); return 0;} |  |

* Lines 0-11 include the needed header files containing the function prototypes and the data structure needed to write the program;
* Lines 15-77 define a function to setup the uinput device including exception handling;
* Line 77-92 define a function to close and clear the uinput device when the program is finished;
* Lines 92-113 define a function to send key events using uinput, which is similar to what you saw in the last challenge session;
* Lines 113-138 define useful variables and set up the uinput device and “hcsr04” sensor with exception handling;
* Lines 138-170 indicate an infinite loop, which keeps writing and reading from the sensor every 0.3 seconds; the distance measured is classified into 3 kinds, those greater than 50 are ignored (meaning your hand is not present nearby, and hence no action); those between 15 and 50 will send 4 successive KEY\_LEFT presses; those smaller than 15 will send 4 successive KEY\_RIGHT presses;
* Lines 172-175 will free and close the uinput device and the “hcsr04” sensor (although these will never be executed in the example program).

When the program is finished, build the program on your host machine (remember you need g++):

arm-linux-gnueabihf-g++ -o test\_hsr04 test\_hcsr04.c

Then, the executable can be either copied to the Micro SD card directly, via a USB stick, or using scp.

Now you can pre-test your program; however, it’s not perfect. If you look back to the kernel module provided in the main lab that enables the hcsr04 sensor, in the hcsr04\_write function, there are two while loops that keep listening the value from GPIO\_IN. In the first while loop, if a pulse is sent but never reflected or reflected with non-detectable strength, then the program will stay in the while loop forever. Although this happens occasionally, if we want to use it as a game controller, then constant reading and writing may well trigger the bug frequently.

To overcome this issue, we shall borrow the concept of a watchdog timer that could bypass the while loop after a certain amount of time. The approach is simply replacing the while loop with a for loop (in the ***hcsr04.c*** file):

ssize\_t hcsr04\_write(struct file \*filp, const char \*buffer, size\_t length, loff\_t \* offset)

{

 printk( KERN\_INFO "hcsr04\_dev: %s\n", \_\_func\_\_ );

 gpio\_set\_value( GPIO\_OUT, 1 );

 udelay( 10 );

 gpio\_set\_value( GPIO\_OUT, 0 );

 int i;

 for (i=0; i<1000000; i++)

 {

 if (gpio\_get\_value( GPIO\_IN )==1)

 goto node\_1;

 }

 node\_1: rising = ktime\_get();

 for (i=0; i<1000000; i++)

 {

 if (gpio\_get\_value( GPIO\_IN )==0)

 goto node\_2;

 }

 node\_2: falling = ktime\_get();

 gpio\_set\_value( GPIO\_OUT, 0 );

 return( 1 );

}

Note that the upper limit 1000000 of “i” is only a reference. The idea is having the for loop wait for sufficient time until the pulse is reflected back but not too long for human to inspect. Depending on your processing power, you can modify the number accordingly. Then, copy the above code to replace the hcsr04\_write function in ***hcsr04.c***.

Now build the image - This should be done with the ***local.conf*** file that worked to build ***core-image-sato***, however we should now add:

IMAGE\_INSTALL\_append += "hcsr04"

Then run:

bitbake -c clean core-image-sato

bitbake core-image-sato

Once complete, a new Micro SD card image will be available, which you can deploy to the Micro SD as follows (assuming the Micro SD is available to the PC as /dev/sdN). Alternatively, use a program of your preference to flash the image.

First run the:

sudo fdisk -l

command to determine which device to flash to (plug in and unplug the SD card to determine which device it is). For this example, the SD card is under the name “sdc” (this may be different in your environment). Next, ensure that the device is unmounted. This can be done using the command:

sudo umount /dev/sdc\*

Once this is done, the following command can be used to copy the image across to the SD card (substitute any folder names and device names to ensure they are relevant to your specific environment).

sudo dd bs=1M if=/home/user/raspberryPi3/rpi-build/tmp-glibc/deploy/images/raspberrypi3/core-image-sato-raspberrypi3.rpi-sdimg of=/dev/sdc

Copy the test\_hcsr04 executable across to the board once more, as done earlier in the lab.

Load the kernel module with the following command:

insmod /lib/modules/4.1.21/kernel/drivers/my\_mod/hcsr04.ko

While you are loading the kernel module, make sure you remember the major and minor number of it to create the associated device file.

mknod /dev/hcsr04 c 243 0

Finally, navigate to the folder containing the updated test\_hcsr04 executable and run:

./test\_hcsr04

This should loop the sensor and continually pick up readings - you can now pair this with playing atanks and use it to control the game.

Note: Serial communication (minicom) cannot detect key strokes from the button press/sensor output, and hence the Raspberry Pi 3 board must be connected to a display via Micro HDMI to successfully observe the key events.