DUBLIN CITY UNIVERSITY

ELECTRONIC AND COMPUTER ENGINEERING

EE513 Connected Embedded Systems

Assignment 1

Author

Michael Lenehan michael.lenehan4@mail.dcu.ie

Student Number: 15410402

09/03/2020

Declaration

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. I have read and understood the Assignment Regulations set out in the module documentation. I have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited are identified in the assignment references. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

I have read and understood the DCU Academic Integrity and Plagiarism at `https://www4.dcu.ie/sites/default/files/policy/1%20-%20integrity_and_plagiarism_ovpaa_v3.pdf` and IEEE referencing guidelines found at `https://loop.dcu.ie/mod/url/view.php?id=448779`.

Signed: ___________________ Date: 09/03/2020

Michael Lenehan

Contents

Listings

1 Introduction

This assignment aims to introduce students to integrating physical sensors or devices with an embedded Linux platform. In this case the physical device being used is the DS3231 Real Time Clock module, and the embedded Linux platform is a Raspberry Pi 3 Model B+. The devices can communicate via the I2C protocol.

The requirements of the assignment state that a C++ class must be implemented for this integration. This class must contain all methods required for the reading and writing of the RTC time, date, alarm, control, and temperature registers. The student must also implement a "novel function" of their choosing.

In completing this assignment, the student should have a greater understanding of the object oriented code required for I2C communications, and the code required to read or write data to/from a physical devices registers.

2 Assignment Setup

2.1 RTC Circuit

The following components are used for this assignment:

1. Raspberry Pi 3 Model B+
2. Samsung SD Card
3. Maxim Integrated DS3231 Real Time Clock Breakout Board.

The I2C protocol is used for communications between the RTC and the Raspberry Pi. As such, the RTC must be connected to the Raspberry Pi via the I2C data and clock connections, on pins 3 and 5 (GPIO 2 and 3) respectively. The RTC breakout board operates on a 3.3V supply voltage, and as such the 3.3V (pin 1), and ground (pin 6) pins must also be connected.

Connecting these pins results in the following layout:

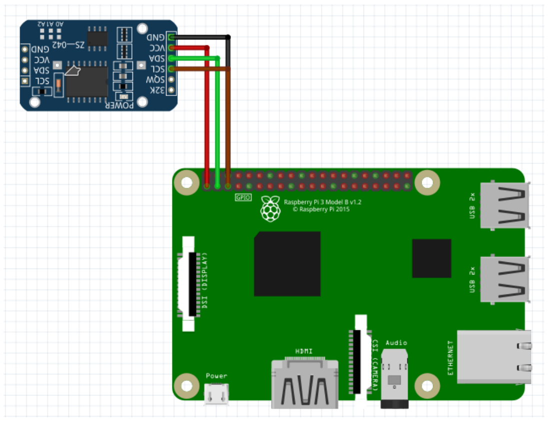


Figure 1: I2C connection from the RTC to the Raspberry Pi

The physical circuit appears as follows:

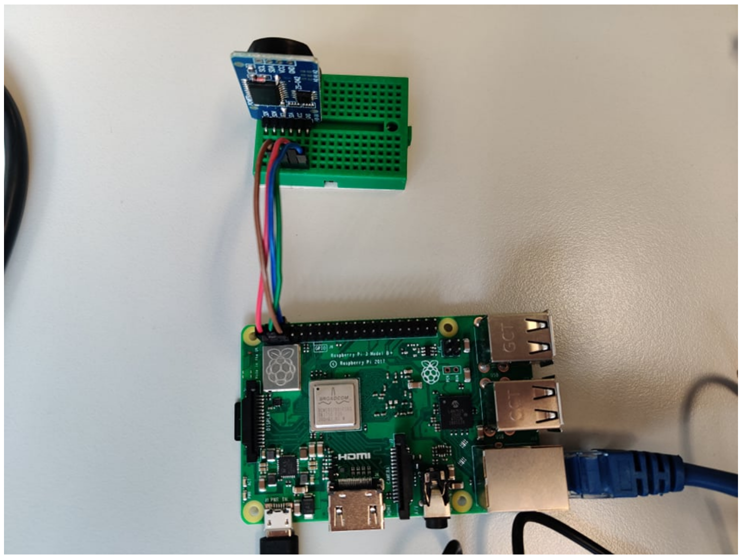


Figure 2: Physical connection from the RTC to the Raspberry Pi

The RTC breakout is connected to a breadboard via the connected male header pins. Four female to male wires connect the Raspberry Pi male GPIO pins to the breadboard. The wire colours in Figure 2 are the same as in Figure 1, with the exception of the ground wire, which in the case of the physical circuit is blue.

2.2 I2C Discovery and Testing

In order to setup I2C on the Raspberry Pi 3 Model B+, the I2C bus must first be enabled. This is achieved by executing the command "raspi-config". Navigating to the "Interfacing Options" tab, and selecting "I2C" to enable the bus.

Once the circuit is connected, the i2ctools package must be installed in order to verify the connection. The package can be installed by executing the following command:

```
sudo apt install i2c-tools
```

Once installed the I2C bus can be scanned using the "i2cdetect command". The "-l" option lists the installed busses.

```
i2cdetect -l
```

The output of this command corresponds to the I2C bus that the device is connected to. Using the "-r" option, bus 1 can be scanned using the SMBus "receive byte" method. The "-y" option in this case disables user input for confirmation.

```
i2cdetect -y -r 1
```

Finally the “i2cdump” command can be used to dump the contents of the specified address, in this case “0x68” on bus 1:

```
i2cdump -y 1 0x68
```

3 C++ Code

In order to develop C++ code for the Raspberry Pi, the code must be compiled via an ARM compatible C++ compiler. This can be achieved in a number of ways, including cross-compilation and compilation on the target hardware (i.e. the Raspberry Pi). As the Raspberry Pi is using the Raspbian “Stretch” image, there is no graphical user environment, i.e. no window manager, or desktop environment. As such, remote development was used to develop on the Raspberry Pi. The “Visual Studio Code” IDE has the ability to connect to a host via SSH, allowing a developer to work on the targets filesystem within the IDE. A terminal can also be opened from within the IDE, allowing terminal commands, including compilation commands, to be run from within the IDE.

Using the provided C code, an I2C base class was created. This class contains all of the generic I2C code required for opening I2C bus connections, device connections, setting read addresses, and reading from registers.

A DS3231 class, which is a child of the I2CDevice class, has a number of methods which are specific to the DS3231. These include all of the required methods for reading and writing time, date, alarm, interrupt, and temperature data to the device. All of the code used for this assignment can be found within the appendices.

3.1 Reading Time and Date

To read the time and date, the I2C bus connection must be opened, the device address must be connected to, the read address must be specified, and the values must be read into a buffer. This functionality was implemented within the "I2CDevice" class. The "setupProc" method calls the "openBus", "connectDevice", "resetReadAddr" and "readBuffer" methods each time an object of the I2CDevice class is created, opening the connection to the device.

```
void I2CDevice::setupProc(){
    this->file = openBus();
    connectDevice();
    resetReadAddr();
    this->readBuffer();
}
```

Listing 1: setupProc Function

From the read buffer, the desired values can be displayed. This functionality is implemented using the "DS3231" class "readTimeAndDate" method:

```
void DS3231::readTimeAndDate(){
    printf("The RTC time is %02d:%02d:%02d\n", bcdToDec(
        buf[2]),
         bcdToDec(buf[1]), bcdToDec(buf[0]));
    printf("The date is %02d - %02d/%02d/%02d\n",
    bcdToDec(buf[3]), bcdToDec(buf[4]), bcdToDec(buf[5]),
         bcdToDec(buf[6]));
    return;
}
```

Listing 2: readTimeAndDate Function

This method reads from the registers indexed by values '2', '1', and '0', which are the hours, minutes, and seconds registers respectively, printing their values in decimal form. As the values read from these registers are in binary coded decimal format, the provided “bcdToDec” function is used to convert them. The date values are read from registers ‘4’, ‘5’, and ‘6’, which are the day, month and year registers respectively. These values are also stored in BCD format, and so the “bcdToDec” function must be used for the purpose of printing.

3.2 Reading Temperature

The temperature value can be read from the RTC in much the same way as the time, however, the values within the temperature registers are stored in two’s compliment representation, with the upper two bits of the least significant bits register representing a ratio of the degree value. With a precision of ± 0.25 degrees, the LSB bits can represent a 0.00 degrees (bits 00), 0.25 degrees (bits 01), 0.50 degrees (bits 10), and 0.75 degrees (bits 11).

```
void DS3231::readTemp(){
    int tempFrac = 100 * (buf[18] >> 6)/4; // Shift to
        extract upper 2 bits, temp is a frac of 100.
    printf("The Temperature is %02d.%02d\n", buf[17],
        tempFrac);
    return;
}
```

Listing 3: readTemp Function

In order to extract the upper two bits of a byte, a bitwise right shift by 6 places is used. Multiplying 100 by the new value and dividing by 4 gives the degrees ratio value. This can then be printed to the screen. The temperature MSB value is stored in the register indexed ‘17’ with the LSB in the register indexed ‘18’.

3.3 Setting Time and Date

In order to write values to the RTC, a write function must be implemented. As the write procedure is generic to I2C devices, this function was added to the "I2CDevice" class.

```
void I2CDevice::writeToReg(unsigned int reg, char val){
    unsigned char writeBuffer[] = {reg, val};
    if(write(file, writeBuffer, 2)!=2){
        perror("Failed to write\n");
        return;
    }
}
```

Listing 4: writeToReg Function

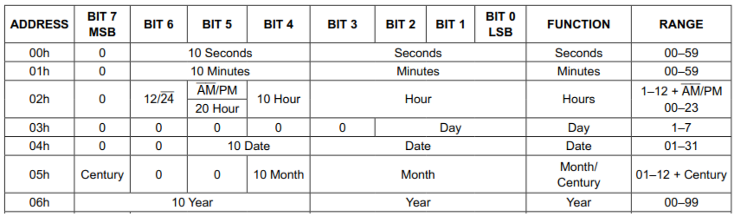
This function takes a register number and an input value. The register value is then used as the address at which to begin writing the new value.

For the DS3231, the time and date must be written in Binary Coded Decimal. As such a "decToBCD" function was implemented. This function takes an input character representing a decimal value, and outputs a BCD encoded value.

```
int DS3231::decToBCD(char b){ return (b/10)*16 + (b%10);
    }
```

Listing 5: decToBCD Function

The time values can then be written to the corresponding registers. The time and date registers are as described above. The table in Figure 3 shows the bits for each of the registers, and the functions of each bit.



ADDRESS	BIT 7 MSB	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0 LSB	FUNCTION	RANGE
00h	0	10 Seconds			Seconds				Seconds	00–59
01h	0	10 Minutes			Minutes				Minutes	00–59
02h	0	12/$\overline{24}$	AM/$\overline{\text{PM}}$ / 20 Hour	10 Hour	Hour				Hours	1–12 + AM/$\overline{\text{PM}}$ 00–23
03h	0	0	0	0	0	Day			Day	1–7
04h	0	0	10 Date		Date				Date	01–31
05h	Century	0	0	10 Month	Month				Month/ Century	01–12 + Century
06h	10 Year				Year				Year	00–99

Figure 3: Time and Date Register Layout

The code in Listing 6 shows the "writeTime" and "writeDate" functions, which pass the input character array values to the "writeToReg" function in order to be written to the RTC. As the upper nibble of the "Hours" register contain control bits, these bits must not be modified during a write operation. The "hrRegChangeUpperBits" function modifies only the LSB of the upper nibble. The "Dates" register contains a "10 Date" value, to which a value is assigned using the "dateChangeUpperBits" function. A check is used within the "writeDate" function to test if the date value is within the appropriate range.

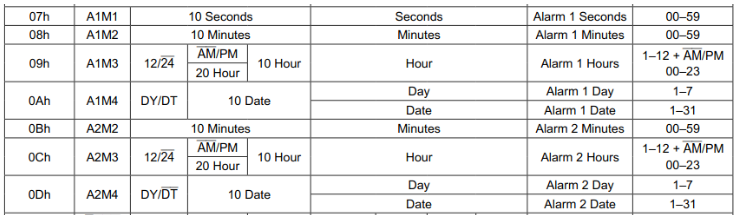
```
void DS3231::writeTime(char* times){
    writeToReg(SEC_REG, decToBCD(times[0]));
    writeToReg(MIN_REG, decToBCD(times[1]));
    writeToReg(HR_REG, decToBCD(hrRegChangeUpperBits(
        times[2], 2)));
    return;
}

void DS3231::writeDate(char* date){
    if((date[0] > 0 && date[0] < 8) && date[1] <= 31){
        writeToReg(DAY_REG, decToBCD(date[0]));
        char dateVal = dateRegChangeUpperBits(date[1], 4)
            ;
        writeToReg(DATE_REG, decToBCD(dateVal));
        writeToReg(MTH_REG, decToBCD(date[2]));
        writeToReg(YR_REG, decToBCD(date[3]));
    }
    else perror("Incorrect Date Entered\n");
    return;
}
```

Listing 6: writeTime and writeDate Functions

3.4 Setting Alarms and Interrupts

Reading and setting the alarms works in much the same way as reading and writing the time, and the alarm values are also stored in BCD encoded format. In order to read the value of the alarms, the values in registers 7, 8, 9 and 10 can be printed. For alarm 2 the values in registers 11, 12 and 13 can be printed. As these are stored in BCD format, the bcdToDec function needs to be called.

Writing the alarms works in a similar manner to writing the time and date, however as the upper bits of each register contains control bits, care must be taken not to unintentionally overwrite the values within these positions. Again, the decToBCD function is required to be used to convert the values to binary coded decimal. Should this function not be called, the alarm values will not match the time values, and the interrupt cannot be triggered. A bitwise "clear" operation is used in order to clear the "Alarm Mask" bits. This allows the interrupt condition to be due to a match in date, seconds, minutes, and hours.



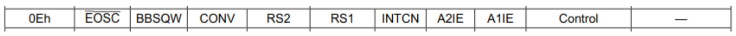
07h	A1M1	10 Seconds			Seconds	Alarm 1 Seconds	00–59
08h	A1M2	10 Minutes			Minutes	Alarm 1 Minutes	00–59
09h	A1M3	12/$\overline{24}$	$\overline{\text{AM}}$/PM 20 Hour	10 Hour	Hour	Alarm 1 Hours	1–12 + $\overline{\text{AM}}$/PM 00–23
0Ah	A1M4	DY/DT	10 Date		Day	Alarm 1 Day	1–7
					Date	Alarm 1 Date	1–31
0Bh	A2M2	10 Minutes			Minutes	Alarm 2 Minutes	00–59
0Ch	A2M3	12/$\overline{24}$	$\overline{\text{AM}}$/PM 20 Hour	10 Hour	Hour	Alarm 2 Hours	1–12 + $\overline{\text{AM}}$/PM 00–23
0Dh	A2M4	DY/$\overline{\text{DT}}$	10 Date		Day	Alarm 2 Day	1–7
					Date	Alarm 2 Date	1–31

Figure 4: Alarm 1 and Alarm 2 Registers

```
void DS3231::setAlarms(char* times){
    writeToReg(A1_SEC, decToBCD((times[0] &= ~(1UL << 7))
        ));
    writeToReg(A1_MIN, decToBCD((times[1] &= ~(1UL << 7))
        ));
    char hr = hrRegChangeUpperBits(times[2], 9);
    hr &= ~(1UL << 7);
    writeToReg(A1_HR, decToBCD(hr));
    char dy = dateRegChangeUpperBits(times[3], 10);
    dy &= ~(1UL << 7);
    writeToReg(A1_DAY, decToBCD(dy));
    writeToReg(A2_MIN, decToBCD((times[4]) &= ~(1UL << 7)
        ));
    char hr2 = hrRegChangeUpperBits(times[5], 12);
    hr2 &= ~(1UL << 7);
    writeToReg(A2_HR, decToBCD(hr2));
    char dy2 = dateRegChangeUpperBits(times[6], 13);
    dy2 &= ~(1UL << 7);
    writeToReg(A2_DAY, decToBCD(dy2));
    return;
}
```

Listing 7: writeAlarm Function

The interrupt can be set by setting the 3rd bit of the control register high. This bit controls the square wave generator output, and, as such, setting the bit high will disable the output when the interrupt alarm condition is met. The “setInterrupt” function takes two boolean inputs, which are used to confirm setting the “A1E” and “A2E” alarm enable bits. If either boolean is true, the interrupt bit is set high, along with the bit of the corresponding alarm. If neither boolean is true, the interrupt bit is set low. Once all of the required bits are set/cleared using the “changeBits” function, the value within the buffer can be written to the register.



0Eh	$\overline{\text{EOSC}}$	BBSQW	CONV	RS2	RS1	INTCN	A2IE	A1IE	Control	—

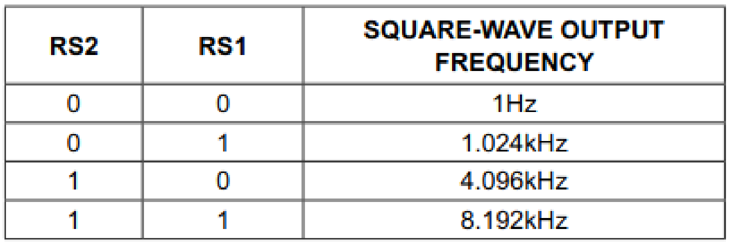
Figure 5: Control Register

```
void DS3231::setInterrupt(bool alarm1, bool alarm2){
    if(alarm1 || alarm2){
        changeBits(true, 14, 2);
        if(alarm1){ changeBits(true, 14, 0); }
        else{ changeBits(false, 14, 0); }
        if(alarm2){ changeBits(true, 14, 1); }
        else{ changeBits(false, 14, 1); }

    }
    else{
        changeBits(false, 14, 2);
    }
    writeToReg(CNTL, buf[14]);
    return;
}
```

Listing 8: set Interrupt Function

3.5 Square Wave Generator

The square wave generator functionality of the RTC can be utilised when the interrupt bit is set low. The output of the square wave generator can be one of four predetermined values, as indicated in Figure 6.



RS2	RS1	SQUARE-WAVE OUTPUT FREQUENCY
0	0	1Hz
0	1	1.024kHz
1	0	4.096kHz
1	1	8.192kHz

Figure 6: Square Wave Generator Frequency

The "sqWaveGen" function, shown in Listing 9 was implemented in order to set the frequency of the square wave generator output. This function takes an integer value input, representing the four available frequency outputs, and sets the values of the Rate Select bits based on this input.

```
void DS3231::sqWaveGen(int freqOption, bool stopInterrupt
    ){
    if(stopInterrupt){
        changeBits(false, 14, 2);
    }
    switch(freqOption){
        case 1:
            changeBits(false, 14, 3);
            changeBits(false, 14, 4);
            break;
        case 2:
            changeBits(true, 14, 3);
            changeBits(false, 14, 4);
            break;
        case 3:
            changeBits(false, 14, 3);
            changeBits(true, 14, 4);
            break;
        case 4:
            changeBits(true, 14, 3);
            changeBits(true, 14, 4);
            break;
        default:
            printf("Invalid Option\n");
            break;
    }
    writeToReg(CNTL, buf[14]);
    printf("%02d\n", buf[14]);
    return;
}
```

Listing 9: sqWaveGen Function

A switch statement is used in order to set or clear the bits in the desired positions, using the changeBits function. Once the switch statement exits, the writeToReg function is used to write the new value in the buffer to the register.

3.6 Novel Functionality

For the novel functionality, it was decided that it would be a useful feature to be able to initialize the RTC time based on the current system time. This was done using a combination of file I/O operations, and a system call to the Linux “date” utility. This utility allows for the formatting of the output based on a number of input arguments. In this case the arguments “+%S %M %H %u %d %m %Y” specify to output the seconds, minutes, hours, weekday number, date, month, and year, delimited by spaces.

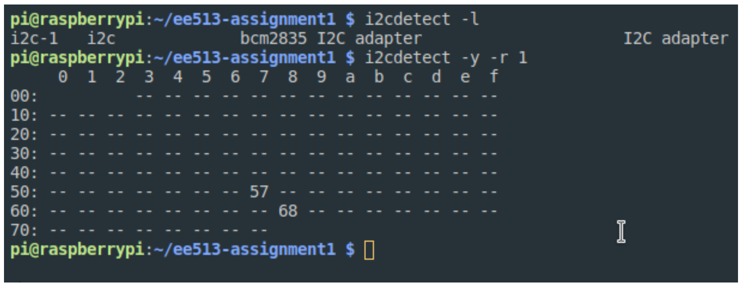
The output of the “date” command is passed into a temporary file, “tmp” from which it can be read, with the symbols delimited by the whitespace. The values are placed in a string vector via the “push_back” method. The temporary file is then closed and deleted. The vector values are converted to integers using the “stoi” function, and placed in separate time and date character arrays, from which they can be passed into the “writeTime” and “writeDate” functions.

```
void DS3231::sysTimeRTCInit(){
    std::string line;
    std::vector<std::string> tmpVals;
    std::ofstream outfile ("tmp");
    outfile.close();
    std::ifstream infile("tmp");
    system("date '+%S %M %H %u %d %m %Y'>> tmp");
    if(infile.is_open()){
        while(getline(infile, line, ' ')){
            tmpVals.push_back(line);
        }
    }
    infile.close();
    remove("tmp");
    char time[3];
    char date[4];
    for(int i=0; i<3; i++){
        time[i] = stoi(tmpVals[i]);
    }
    for(int i=0; i<3; i++){
        date[i] = stoi(tmpVals[i+3]);
    }
    date[3] = stoi(tmpVals[6].substr(2,3));
    writeTime(time);
    writeDate(date);
}
```

Listing 10: Novel Function - Initialize RTC time from System Time

4 Testing

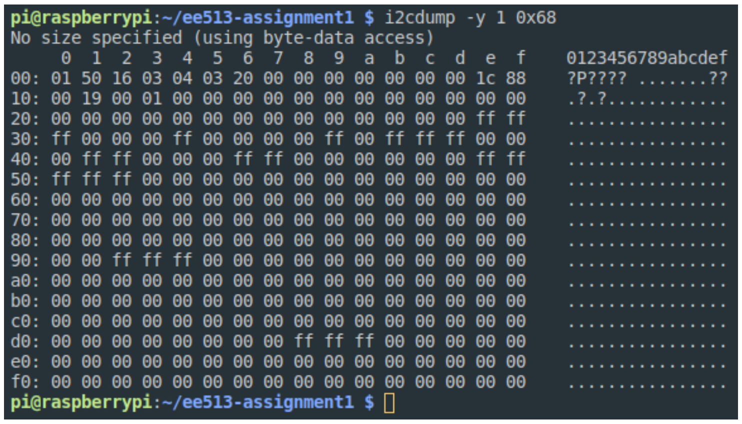
The “i2ctools” detect commands listed in Section 2.2 were used to list the connected I2C busses, and to show the devices connected to the busses. From the screenshot shown in Figure 7 it can be seen that the RTC was installed on bus 1 (i2c-1), and had an address of 0x68.



```
pi@raspberrypi:~/ee513-assignment1 $ i2cdetect -l
i2c-1   i2c             bcm2835 I2C adapter                     I2C adapter
pi@raspberrypi:~/ee513-assignment1 $ i2cdetect -y -r 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- 57 -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- 68 -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
pi@raspberrypi:~/ee513-assignment1 $
```

Figure 7: I2C Detect Output

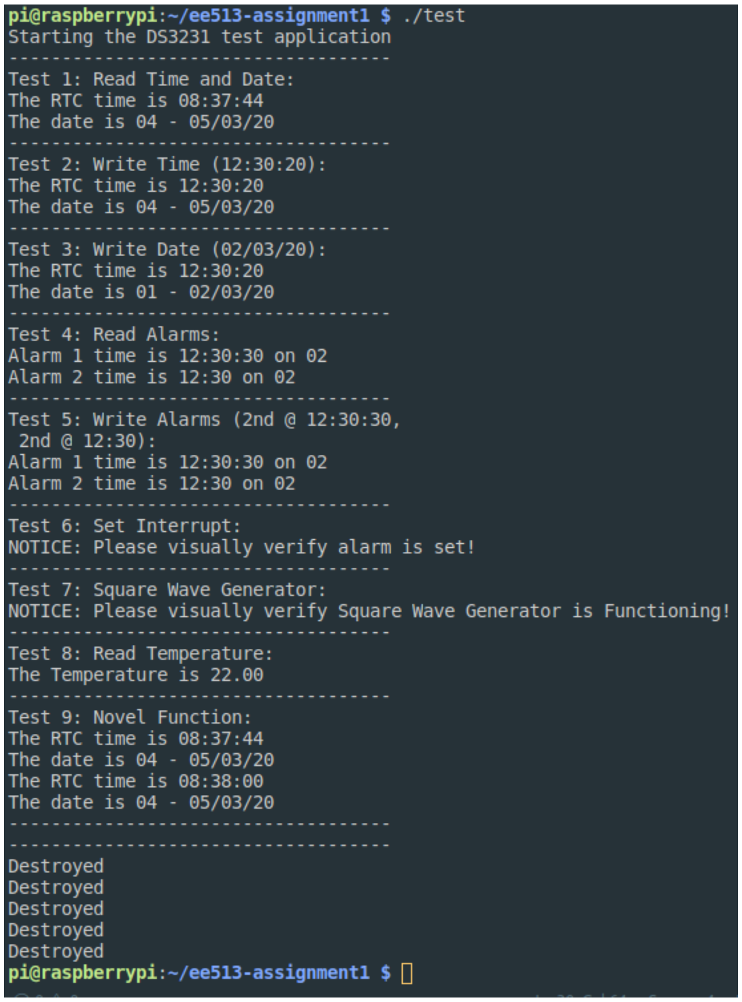
The dump command, also listed in Section 2.2 was used to show the values stored in the RTC registers. Registers 0x00 to 0x12 contain the time, date, alarm, temperature and control registers. The values contained in these registers can be seen in Figure 8.



```
pi@raspberrypi:~/ee513-assignment1 $ i2cdump -y 1 0x68
No size specified (using byte-data access)
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f    0123456789abcdef
00: 01 50 16 03 04 03 20 00 00 00 00 00 00 00 1c 88    ?P???? .......??
10: 00 19 00 01 00 00 00 00 00 00 00 00 00 00 00 00    .?.?............
20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ff ff    ................
30: ff 00 00 00 ff 00 00 00 00 ff 00 ff ff ff 00 00    ................
40: 00 ff ff 00 00 00 ff ff 00 00 00 00 00 00 ff ff    ................
50: ff ff ff 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
90: 00 00 ff ff ff 00 00 00 00 00 00 00 00 00 00 00    ................
a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
d0: 00 00 00 00 00 00 00 00 ff ff ff 00 00 00 00 00    ................
e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
pi@raspberrypi:~/ee513-assignment1 $
```

Figure 8: I2C Dump Output

In order to verify the execution of the code, a test script was written. This script contains the necessary function calls to test the required functionality for this assignment. Before executing the test executable, the novel function is run (using the main.cpp file). This sets the time on the RTC to the initial value read in Figure 9.



```
pi@raspberrypi:~/ee513-assignment1 $ ./test
Starting the DS3231 test application
------------------------------------
Test 1: Read Time and Date:
The RTC time is 08:37:44
The date is 04 - 05/03/20
------------------------------------
Test 2: Write Time (12:30:20):
The RTC time is 12:30:20
The date is 04 - 05/03/20
------------------------------------
Test 3: Write Date (02/03/20):
The RTC time is 12:30:20
The date is 01 - 02/03/20
------------------------------------
Test 4: Read Alarms:
Alarm 1 time is 12:30:30 on 02
Alarm 2 time is 12:30 on 02
------------------------------------
Test 5: Write Alarms (2nd @ 12:30:30,
 2nd @ 12:30):
Alarm 1 time is 12:30:30 on 02
Alarm 2 time is 12:30 on 02
------------------------------------
Test 6: Set Interrupt:
NOTICE: Please visually verify alarm is set!
------------------------------------
Test 7: Square Wave Generator:
NOTICE: Please visually verify Square Wave Generator is Functioning!
------------------------------------
Test 8: Read Temperature:
The Temperature is 22.00
------------------------------------
Test 9: Novel Function:
The RTC time is 08:37:44
The date is 04 - 05/03/20
The RTC time is 08:38:00
The date is 04 - 05/03/20
------------------------------------
------------------------------------
Destroyed
Destroyed
Destroyed
Destroyed
Destroyed
pi@raspberrypi:~/ee513-assignment1 $ 
```

Figure 9: Testing Output

The test scripts divides the required functionality into a number of tests. One “DS3231” object executes all of the write functions, with a new object required for each of the subsequent read functions. This is due to the read function reading from the initially stored buffer values.

4.0.1 Test 1: Read Time and Date

As mentioned previously, the initial time read from the RTC was set via the novel function i.e. the system clock time. The output of this test shows the time and date are being read correctly from the RTC.

4.0.2 Test 2: Write Time

The time being written to the registers is 12:30:20, i.e. 12 hours, 30 minutes, and 20 seconds. The read function verifies that this time is being correctly written to the registers. An hour value of greater than 9 was used to verify that the upper nibble of data was not being unintentionally modified, with only the desired bits being changed.

4.0.3 Test 3: Write Date

The date being written to the registers is 02/03/2020, with the day of the week set to 1. Again the read function is used to verify this change. It can be seen from Figure 9 that the date has changed to the input value.

4.0.4 Test 4: Read Alarms

The read alarm function output shows the values which are stored in the alarm registers. These values are set from a previous execution of the test script.

4.0.5 Test 5: Write Alarms

The time and date being written to the alarm registers are 12:30:30 on the 2nd for Alarm 1 and 12:30 on the 2nd for Alarm 2. As these values were written on a previous execution, there is no change to the register, as is expected.

4.0.6 Test 6: Set Interrupt

The set interrupt function allows for the alarm flag to trigger the interrupt on the square wave output. A delay in the code allows the user to verify that the interrupt is triggered. The alarm flag being raised causes the interrupt output to stop, i.e. the LED being used for visual verification turning off. This can be seen in the following images, the first showing that the LED is lit before the alarm condition is met, and the second showing the LED is not lit after the alarm condition is met.

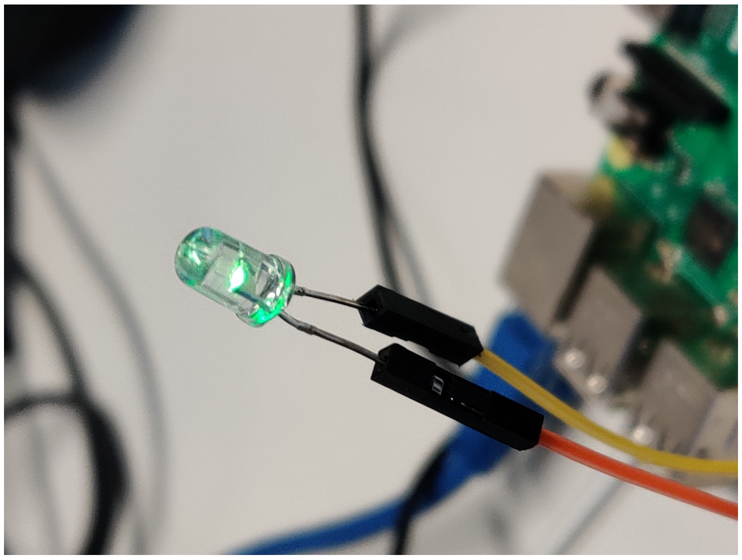


Figure 10: LED before alarm condition

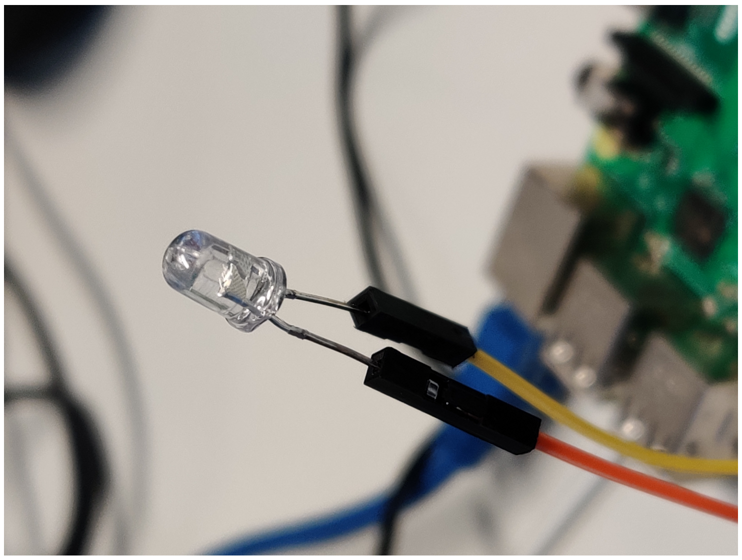


Figure 11: LED after alarm condition

4.0.7 Test 7: Square Wave Generator

The Square Wave Generator function was written with a value of “1”, representing a frequency of 1Hz. Again this was visually verified. The flashing LED cannot be demonstrated in the report.

4.0.8 Test 8: Read Temperature

The read temperature function prints the current temperature on the RTC, with a precision of $\pm 0.25°C$. This test outputs a value of 22°C.

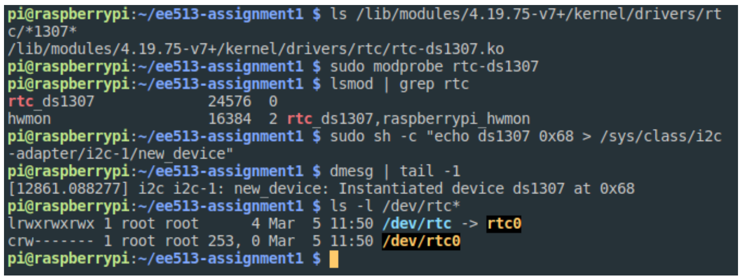
4.0.9 Test 9: Novel Function

The novel function sets the RTC time and date to the system time and date, as output from the Linux “date” utility. This was verified, as the output time and date, as in

Figure 9, were the current time and date of execution of the program.

5 Linux Kernel Module

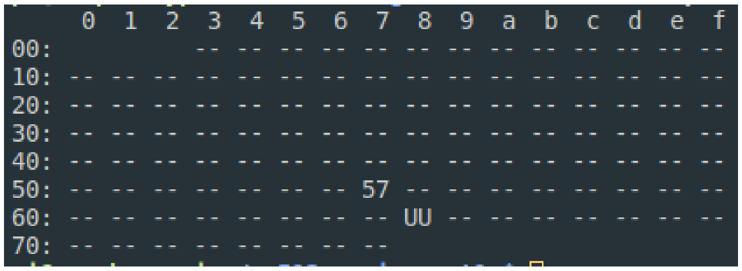
This section of the assignment implements a Linux Kernel Module in order to read and write the RTC time. The “rtc-ds1307.ko” LKM is compatible with the DS3231, and will be used for this section. The Linux “modprobe” program is used to add the LKM to the kernel. The ds1307 device is then instantiated by writing its address to the i2c-1 directories “new_device” file. This process can be seen in Figure 12.



```
pi@raspberrypi:~/ee513-assignment1 $ ls /lib/modules/4.19.75-v7+/kernel/drivers/rt
c/*1307*
/lib/modules/4.19.75-v7+/kernel/drivers/rtc/rtc-ds1307.ko
pi@raspberrypi:~/ee513-assignment1 $ sudo modprobe rtc-ds1307
pi@raspberrypi:~/ee513-assignment1 $ lsmod | grep rtc
rtc_ds1307             24576  0
hwmon                  16384  2 rtc_ds1307,raspberrypi_hwmon
pi@raspberrypi:~/ee513-assignment1 $ sudo sh -c "echo ds1307 0x68 > /sys/class/i2c
-adapter/i2c-1/new_device"
pi@raspberrypi:~/ee513-assignment1 $ dmesg | tail -1
[12861.088277] i2c i2c-1: new_device: Instantiated device ds1307 at 0x68
pi@raspberrypi:~/ee513-assignment1 $ ls -l /dev/rtc*
lrwxrwxrwx 1 root root      4 Mar  5 11:50 /dev/rtc -> rtc0
crw------- 1 root root 253, 0 Mar  5 11:50 /dev/rtc0
pi@raspberrypi:~/ee513-assignment1 $ 
```

Figure 12: Adding the DS1307 LKM to the Kernel

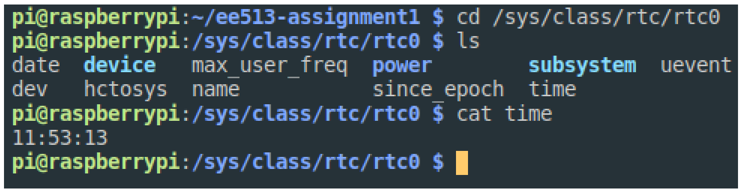
Executing the “i2cdetect” command shows a value of “UU” at the RTC’s address, showing the device is in use by an LKM.



```
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- 57 -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- UU -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Figure 13: i2cdetect Output

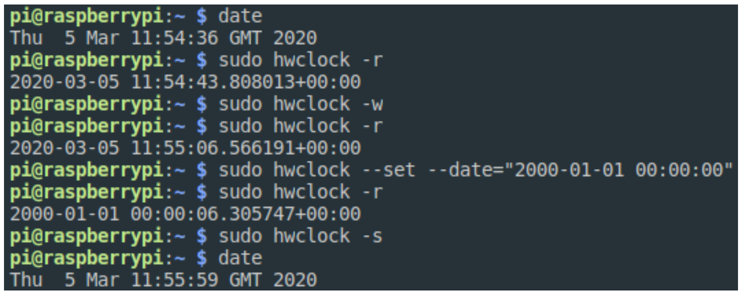
By navigating to the devices “sysfs” entry, the “cat” command can be used to output the current RTC time to the user.



```
pi@raspberrypi:~/ee513-assignment1 $ cd /sys/class/rtc/rtc0
pi@raspberrypi:/sys/class/rtc/rtc0 $ ls
date  device   max_user_freq  power        subsystem  uevent
dev   hctosys  name           since_epoch  time
pi@raspberrypi:/sys/class/rtc/rtc0 $ cat time
11:53:13
pi@raspberrypi:/sys/class/rtc/rtc0 $ 
```

Figure 14: RTC output via the “cat” command

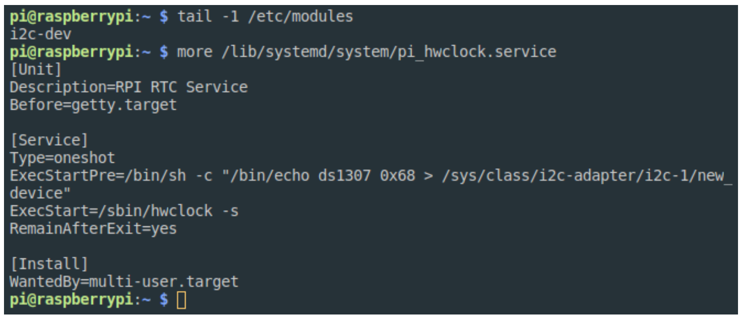
The “hwclock” utility can also be used to read or write to the RTC, with the “set” option allowing for the system time to be set from the RTC.



```
pi@raspberrypi:~ $ date
Thu  5 Mar 11:54:36 GMT 2020
pi@raspberrypi:~ $ sudo hwclock -r
2020-03-05 11:54:43.808013+00:00
pi@raspberrypi:~ $ sudo hwclock -w
pi@raspberrypi:~ $ sudo hwclock -r
2020-03-05 11:55:06.566191+00:00
pi@raspberrypi:~ $ sudo hwclock --set --date="2000-01-01 00:00:00"
pi@raspberrypi:~ $ sudo hwclock -r
2000-01-01 00:00:06.305747+00:00
pi@raspberrypi:~ $ sudo hwclock -s
pi@raspberrypi:~ $ date
Thu  5 Mar 11:55:59 GMT 2020
```

Figure 15: “hwclock” utility usage

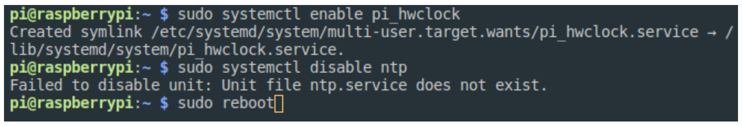
To automatically set the time via the RTC at boot time, a systemd service can be added to the “system” directory. The contents of this service file can be seen in Figure 16.



```
pi@raspberrypi:~ $ tail -1 /etc/modules
i2c-dev
pi@raspberrypi:~ $ more /lib/systemd/system/pi_hwclock.service
[Unit]
Description=RPI RTC Service
Before=getty.target

[Service]
Type=oneshot
ExecStartPre=/bin/sh -c "/bin/echo ds1307 0x68 > /sys/class/i2c-adapter/i2c-1/new_
device"
ExecStart=/sbin/hwclock -s
RemainAfterExit=yes

[Install]
WantedBy=multi-user.target
pi@raspberrypi:~ $ 
```

Figure 16: System service integration

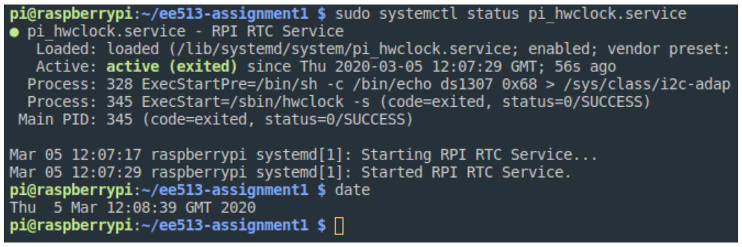
In order to start this service, the “systemctl” command can be run. The NTP service must also be disabled, however, as shown in Figure 17, the “ntp” service is not available. This is because it is no longer included in the Raspbian Stretch images.



```
pi@raspberrypi:~ $ sudo systemctl enable pi_hwclock
Created symlink /etc/systemd/system/multi-user.target.wants/pi_hwclock.service → /
lib/systemd/system/pi_hwclock.service.
pi@raspberrypi:~ $ sudo systemctl disable ntp
Failed to disable unit: Unit file ntp.service does not exist.
pi@raspberrypi:~ $ sudo reboot
```

Figure 17: Enabling the system service

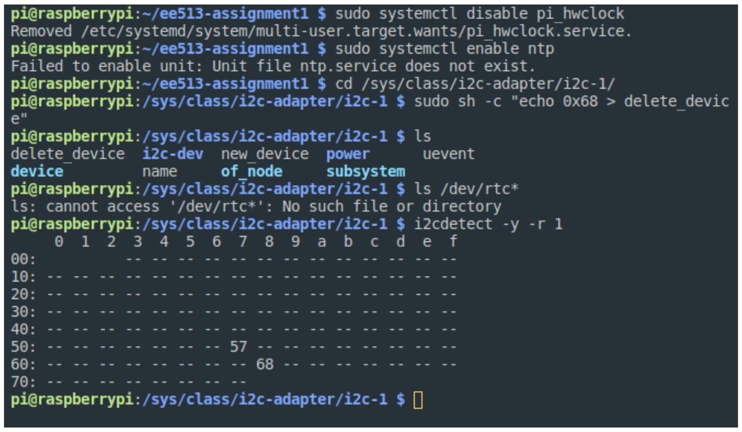
Once the system has been rebooted, the status of this system service can be checked. Figure 18 shows that the service is active after a reboot.



```
pi@raspberrypi:~/ee513-assignment1 $ sudo systemctl status pi_hwclock.service
● pi_hwclock.service - RPI RTC Service
   Loaded: loaded (/lib/systemd/system/pi_hwclock.service; enabled; vendor preset:
   Active: active (exited) since Thu 2020-03-05 12:07:29 GMT; 56s ago
  Process: 328 ExecStartPre=/bin/sh -c /bin/echo ds1307 0x68 > /sys/class/i2c-adap
  Process: 345 ExecStart=/sbin/hwclock -s (code=exited, status=0/SUCCESS)
 Main PID: 345 (code=exited, status=0/SUCCESS)

Mar 05 12:07:17 raspberrypi systemd[1]: Starting RPI RTC Service...
Mar 05 12:07:29 raspberrypi systemd[1]: Started RPI RTC Service.
pi@raspberrypi:~/ee513-assignment1 $ date
Thu  5 Mar 12:08:39 GMT 2020
pi@raspberrypi:~/ee513-assignment1 $ 
```

Figure 18: System service status

Finally, in order to remove the LKM and undo the creation of the system service, the service must be disabled, with the ntp service re-enabled. As mentioned, this service no longer exists within the Raspbian Stretch images. The LKM can then be removed by passing the RTC address (0x68) to the “delete_device” file of the “i2c-1” directory. This can be verified using the “i2cdetect” command, which shows the address 0x68 for the RTC, indicating it is no longer in use by the LKM.



```
pi@raspberrypi:~/ee513-assignment1 $ sudo systemctl disable pi_hwclock
Removed /etc/systemd/system/multi-user.target.wants/pi_hwclock.service.
pi@raspberrypi:~/ee513-assignment1 $ sudo systemctl enable ntp
Failed to enable unit: Unit file ntp.service does not exist.
pi@raspberrypi:~/ee513-assignment1 $ cd /sys/class/i2c-adapter/i2c-1/
pi@raspberrypi:/sys/class/i2c-adapter/i2c-1 $ sudo sh -c "echo 0x68 > delete_devic
e"
pi@raspberrypi:/sys/class/i2c-adapter/i2c-1 $ ls
delete_device  i2c-dev  new_device  power      uevent
device         name     of_node     subsystem
pi@raspberrypi:/sys/class/i2c-adapter/i2c-1 $ ls /dev/rtc*
ls: cannot access '/dev/rtc*': No such file or directory
pi@raspberrypi:/sys/class/i2c-adapter/i2c-1 $ i2cdetect -y -r 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- 57 -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- 68 -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
pi@raspberrypi:/sys/class/i2c-adapter/i2c-1 $ 
```

Figure 19: Removing the LKM and freeing the RTC

6 Conclusion

Each of the required read and write functions of the assignment were implemented. The novel functionality was also implemented. Documentation was created using “Doxygen” and can be found in the submit git repository.

The interrupt did not operate as expected. Reading the datasheet indicates that the interrupt bit being set high while the alarm flag is low would result in no output on the square wave output, however the exact opposite is true, with no output on the pin only if the alarm flag is high.

Overall a lot was learned with regards to integrating hardware with an embedded Linux system, including how to communicate via I2C, and how to perform integration testing.

A Appendix

A.1 DS3231.c - Provided

```
#include<stdio.h>
#include<fcntl.h>
#include<sys/ioctl.h>
#include<linux/i2c.h>
#include<linux/i2c-dev.h>
#define BUFFER_SIZE 19      //0x00 to 0x12

// the time is in the registers in encoded decimal form
int bcdToDec(char b) { return (b/16)*10 + (b%16); }

int main(){
   int file;
   printf("Starting the DS3231 test application\n");
   if((file=open("/dev/i2c-1", O_RDWR)) < 0){
      perror("failed to open the bus\n");
      return 1;
   }
   if(ioctl(file, I2C_SLAVE, 0x68) < 0){
      perror("Failed to connect to the sensor\n");
      return 1;
   }
   char writeBuffer[1] = {0x00};
   if(write(file, writeBuffer, 1)!=1){
      perror("Failed to reset the read address\n");
      return 1;
   }
   char buf[BUFFER_SIZE];
```

```
    if(read(file, buf, BUFFER_SIZE)!=BUFFER_SIZE){
        perror("Failed to read in the buffer\n");
        return 1;
    }
    printf("The RTC time is %02d:%02d:%02d\n", bcdToDec(
        buf[2]),
            bcdToDec(buf[1]), bcdToDec(buf[0]));
    close(file);
    return 0;
}
```

A.2 I2CDevice.h

```
/**
 * I2CDevice Class
 */

#include<unistd.h>
#include<stdio.h>
#include<fcntl.h>
#include<sys/ioctl.h>
#include<linux/i2c.h>
#include<linux/i2c-dev.h>
#include<vector>

class I2CDevice{
     protected:
     int file;
     std::vector<char> buf;
     char addr;
```

```
    int openBus();
    void connectDevice();
    void resetReadAddr();
    char* readBuffer();
    void writeToReg(unsigned int, char);
    void setupProc();
    public:
    I2CDevice(int);
    I2CDevice(int, char);
    virtual ~I2CDevice();
};
```

A.3 I2CDevice.cpp

```
#include "I2CDevice.h"

I2CDevice::I2CDevice(int bufSize) : buf(bufSize){
    addr = 0x00;
    setupProc();
}

I2CDevice::I2CDevice(int bufSize, char address) :
    buf(bufSize), addr(address)
{
    setupProc();
}

I2CDevice::~I2CDevice(){
    close(file);
    printf("Destroyed\n");
```

```
}

int I2CDevice::openBus(){

    /** \brief Open the Bus connection to the I2C Device
     *
     */

    int file;
    if((file=open("/dev/i2c-1", O_RDWR)) < 0){
        perror("failed to open the bus\n");
        return 1;
    }
    return file;
}

void I2CDevice::connectDevice(){

    /** \brief Open the connection to the I2C Device
     *
     */

    if(ioctl(file, I2C_SLAVE, 0x68) < 0){
       perror("Failed to connect to the sensor\n");
       return;
    }
}

void I2CDevice::resetReadAddr(){
    /** \brief Set the initial read address for the I2C
        Device
     *
     */

    char writeBuffer[1] = {addr};
    if(write(file, writeBuffer, 1)!=1){
        perror("Failed to reset the read address\n");
        return;
    }
}

char* I2CDevice::readBuffer(){

    /** \brief Reads from the I2C devices registers
     *
     */

    char* buffer = &buf[0];
    if(read(file, buffer, buf.size())!=buf.size()){
        perror("Failed to read in the buffer\n");
        return buffer;
    }
    return buffer;
}

void I2CDevice::writeToReg(unsigned int reg, char val){

    /** \brief Write to the I2C Devices registers
     *
     */

    unsigned char writeBuffer[] = {reg, val};
    if(write(file, writeBuffer, 2)!=2){
        perror("Failed to write\n");
        return;
    }
}

void I2CDevice::setupProc(){

    /** \brief Setup Procedure for I2C Devices
     *
     *  Calls the functions required to open an I2C
        connection to
     * a device, and read the buffers of this device.
     */

    this->file = openBus();
    connectDevice();
    resetReadAddr();
    this->readBuffer();
}
```

A.4 DS3231.h

```
/**
 *  DS3231 Class
 *  Inherits from I2CDevice class
 */
```

```
#ifndef DS3231_h
#define DS3231_h

#include "I2CDevice.h"
#include <stdlib.h>
#include <string>
#include <iostream>
#include <fstream>
#include <cstring>
#define BUFFER_SIZE 19          //0x00 to 0x12
#define SEC_REG 0x00
#define MIN_REG 0x01
#define HR_REG 0x02
#define DAY_REG 0x03
#define DATE_REG 0x04
#define MTH_REG 0x05
#define YR_REG 0x06
#define A1_SEC 0x07
#define A1_MIN 0x08
#define A1_HR 0x09
#define A1_DAY 0x0a
#define A2_MIN 0x0b
#define A2_HR 0x0c
#define A2_DAY 0x0d
#define CNTL 0x0e
#define STAT 0x0f

class DS3231 : public I2CDevice{
    private:
```

```
    int bcdToDec(char);
    int decToBCD(char);
    char hrRegChangeUpperBits(char, int);
    char dateRegChangeUpperBits(char, int);
    char changeBits(bool, int, int);
    public:
    DS3231();
    DS3231(char);
    DS3231(bool);
    void sysTimeRTCInit();
    void readTimeAndDate();
    void writeTime(char*);
    void writeDate(char*);
    void readAlarms();
    void setAlarms(char*);
    void readTemp();
    void setInterrupt(bool, bool);
    void sqWaveGen(int, bool);
    void clearAlarms();
};

#endif
```

A.5 DS3231.cpp

```
#include "DS3231.h"

DS3231::DS3231() : I2CDevice(BUFFER_SIZE){}

```

```
DS3231::DS3231(char address) : I2CDevice(BUFFER_SIZE,
    address){}

DS3231::DS3231(bool setSystemTime) : I2CDevice(
    BUFFER_SIZE){
     if(setSystemTime){
          sysTimeRTCInit();
     }
}

// Converts from bcd to dec - used for reading time/date
    reg
int DS3231::bcdToDec(char b){ return (b/16)*10 + (b%16);
    }

// Converts from dec to bcd - used to write to time/date
    reg
int DS3231::decToBCD(char b){ return (b/10)*16 + (b%10);
    }

char DS3231::hrRegChangeUpperBits(char b, int reg){

     /** \brief Modify the upper nibble of the 1 byte
      *  hour value - Does not affect the 12/24 or
      *  AM/PM bits
      *
      *  This function is used to modify the lower nibble
      *  of the "Hour" register, without unintended
          modifyications
      *  to the bits of the upper nibble.
      *  The 10/20 hour bits are modified as required to
          achieve the
      *  correct 24-hour time representation
      */

     char upperBuffNibble = (this->buf[reg] & 0xF0);
     char upperHrNibbleVal = (b & 0xF0) >> 4;
     if(upperHrNibbleVal == 0){
          upperBuffNibble &= ~(1UL << 4);
          upperBuffNibble &= ~(1UL << 5);
     } else if(upperHrNibbleVal == 1){
          upperBuffNibble |= (1UL << 4);
          upperBuffNibble &= ~(1UL << 5);
     }
     else if(upperHrNibbleVal == 2){
          upperBuffNibble &= ~(1UL << 4);
          upperBuffNibble |= (1UL << 5);
     }
     else{
          perror("Invalid Hour Time Entered\n");
          return '\0';
     }
     char lowerHrNibble = b & 0x0F;
     return upperBuffNibble | lowerHrNibble;
}

char DS3231::dateRegChangeUpperBits(char b, int reg){

     /** \brief Modify the upper nibble of the 1 byte
      *  date value.
      *
      *  This function is used to modify the lower nibble
      *  of the "Date" register, without unintended
          modifyications
      *  to the bits of the upper nibble.
      *  The 10 date bits are modified as required to
          achieve the
      *  correct 24-hour time representation
      */

     char upperBuffNibble = (this->buf[reg] & 0xF0);
     char upperDateNibbleVal = (b & 0xF0) >> 4;
     if(upperDateNibbleVal == 0){
          upperBuffNibble &= ~(1UL << 4);
          upperBuffNibble &= ~(1UL << 5);
     } else if(upperDateNibbleVal == 1){
          upperBuffNibble |= (1UL << 4);
          upperBuffNibble &= ~(1UL << 5);
     } else if(upperDateNibbleVal == 2){
          upperBuffNibble &= ~(1UL << 4);
          upperBuffNibble |= (1UL << 5);
     } else if(upperDateNibbleVal ==3){
          upperBuffNibble |= (1UL << 4);
          upperBuffNibble |= (1UL << 5);
     }
     else{
          perror("Invalid Date Entered\n");
          return '\0';
     }
     char lowerDateNibble = b & 0x0F;
     return upperBuffNibble | lowerDateNibble;
}

char DS3231::changeBits(bool set, int reg, int pos){

    /** \brief Set/Clear a bit in the required position
     *  in the required register.
     *
     *  The boolean "set" value is checked for set/clear
     *  operations. Using bitwise operations, the bit in
     *  position "pos" in register "reg" is either set or
     *  cleared.
     */

    if(set){
        return buf[reg] |= 1UL << pos;
    } else{
        return buf[reg] &= ~(1UL << pos);
    }
}

void DS3231::sysTimeRTCInit(){

    /** \brief Sets the RTC time based on the output of
       the
     *  Linux "date" utility.
     *
     *  This function creates a temporary file, runs the
        command
```

```
 *  line "date" utility – formatting to give time (
    sec min hr)
 *  and date (day, date, month, year), and stores the
      output
 *  in the temporary file. This file is read into a
     string vector
 *  and the temporary file is closed and deleted. The
      time values
 *  and date values are added to separate char arrays
      for writing
 *  to the RTC.
 */

std::string line;
std::vector<std::string> tmpVals;
std::ofstream outfile ("tmp");
outfile.close();
std::ifstream infile("tmp");
system("date '+%S %M %H %u %d %m %Y'>> tmp");
if(infile.is_open()){
    while(getline(infile, line, ' ')){
        tmpVals.push_back(line);
    }
}
infile.close();
remove("tmp");
char time[3];
char date[4];
for(int i=0; i<3; i++){
    time[i] = stoi(tmpVals[i]);
    }
    for(int i=0; i<3; i++){
        date[i] = stoi(tmpVals[i+3]);
    }
    date[3] = stoi(tmpVals[6].substr(2,3));
    writeTime(time);
    writeDate(date);
}

void DS3231::readTimeAndDate(){

    /** \brief Print the RTC Time and Date
     *
     *  Prints the values stored in registers 2, 1, and 0
          (Hours,
     *  Minutes, Seconds), followed by registers 3, 4, 5,
          and 6 (
     *  Day, Date, Month, Year).
     */

    printf("The RTC time is %02d:%02d:%02d\n", bcdToDec(
        buf[2]),
        bcdToDec(buf[1]), bcdToDec(buf[0]));
    printf("The date is %02d - %02d/%02d/%02d\n",
    bcdToDec(buf[3]), bcdToDec(buf[4]), bcdToDec(buf[5]),
        bcdToDec(buf[6]));
    return;
}

void DS3231::writeTime(char* times){
    /**\brief Write the input times to the RTC time
       registers
     *
     *  Takes a character array containing "time" values
        to be
     *  written. Calls the "writeToReg" function to write
         the
     *  required value to the required registers.
     *  Defined values are used to reference the second,
        minute
     *  and hour registers (0x00, 0x01, and 0x02).
     */

    writeToReg(SEC_REG, decToBCD(times[0]));
    writeToReg(MIN_REG, decToBCD(times[1]));
    writeToReg(HR_REG, decToBCD(hrRegChangeUpperBits(
       times[2], 2)));
    return;
}

void DS3231::writeDate(char* date){

    /**\brief Write the input dates to the RTC date
       registers
     *
     *  Takes a character array containing "date" values
        to be
     *  written. Calls the "writeToReg" function to write
         the
     *  required value to the required registers.
     *  Defined values are used to reference the day,
        date, month.
     *  and year registers (0x03, 0x04, 0x05 and 0x06).
     *  A check is performed to ensure the date is within
         the
     *  correct range
     */

    if((date[0] > 0 && date[0] < 8) && date[1] <= 31){
        writeToReg(DAY_REG, decToBCD(date[0]));
        char dateVal = dateRegChangeUpperBits(date[1], 4)
            ;
        writeToReg(DATE_REG, decToBCD(dateVal));
        writeToReg(MTH_REG, decToBCD(date[2]));
        writeToReg(YR_REG, decToBCD(date[3]));
    }
    else perror("Incorrect Date Entered\n");
    return;
}

void DS3231::readAlarms(){

    /** \brief Print the RTC Alarms
     *
     *  Prints the values stored in the Alarm 1 and Alarm
         2
     *  registers (0x07 - 0x0D).
     */

    printf("Alarm 1 time is %02d:%02d:%02d on %02d\n",
    bcdToDec(buf[9]), bcdToDec(buf[8]), bcdToDec(buf[7]),
        bcdToDec(buf[10]));
    printf("Alarm 2 time is %02d:%02d on %02d\n",
    bcdToDec(buf[12]), bcdToDec(buf[11]), bcdToDec(buf
       [13]));
    return;
}

void DS3231::setAlarms(char* times){

    /** \brief Set the RTC Alarm Times/Dates
     *
     *  Writes to the Alarm registers (0x07 - 0x0D). Sets
          all
     *  "Alarm Mask Bits" to 0 - Alarm will trigger when
         date, hours,
     *  minutes and seconds match
     */

    writeToReg(A1_SEC, decToBCD((times[0] &= ~(1UL << 7))
       ));
    writeToReg(A1_MIN, decToBCD((times[1] &= ~(1UL << 7))
       ));
    char hr = hrRegChangeUpperBits(times[2], 9);
    hr &= ~(1UL << 7);
    writeToReg(A1_HR, decToBCD(hr));
    char dy = dateRegChangeUpperBits(times[3], 10);
    dy &= ~(1UL << 7);
    writeToReg(A1_DAY, decToBCD(dy));
    writeToReg(A2_MIN, decToBCD((times[4]) &= ~(1UL << 7)
        ));
    char hr2 = hrRegChangeUpperBits(times[5], 12);
    hr2 &= ~(1UL << 7);
    writeToReg(A2_HR, decToBCD(hr2));
    char dy2 = dateRegChangeUpperBits(times[6], 13);
    dy2 &= ~(1UL << 7);
    writeToReg(A2_DAY, decToBCD(dy2));
    return;
}

void DS3231::readTemp(){

    /** \brief Print the RTC temperature
     *
     *  Temperature has a precision of 0.25 degrees.
     *  The upper part of the temperature is in reg 1
     *  The lower part of the temperature is in tbe
     *  upper two bits of reg 2
    */

    int tempFrac = 100 * (buf[18] >> 6)/4; // Shift to
        extract upper 2 bits, temp is a frac of 100.
    printf("The Temperature is %02d.%02d\n", buf[17],
        tempFrac);
    return;
}

void DS3231::setInterrupt(bool alarm1, bool alarm2){
    /** \brief Set the RTC Alarm Interrupt
     *
     *  Takes two boolean values representing the status
        of
     *  alarms 1 and 2. If either are to be set, the
        interrupt
     *  is set, along with the corresponding alarm enable
         bit.
     *  If neither are set, the interrupt bit is cleared,
         i.e. the
     *  Square Wave Generator is enabled.
     */

    if(alarm1 || alarm2){
        changeBits(true, 14, 2);
        if(alarm1){ changeBits(true, 14, 0); }
        else{ changeBits(false, 14, 0); }
        if(alarm2){ changeBits(true, 14, 1); }
        else{ changeBits(false, 14, 1); }

    }
    else{
        changeBits(false, 14, 2);
    }
    writeToReg(CNTL, buf[14]);
    return;
}

void DS3231::sqWaveGen(int freqOption, bool stopInterrupt
    ){
```

```
/** \brief Enable the Square Wave Generator at the
    desired
 *  frequency
 *
 *  Takes an integer value representing one of the
     four
 *  available frequencies of the square wave
     generator.
 *  A boolean represents the toggling of the
     interrupt bit,
 *  which enables the generator is set low.
 *  Each case of the switch statement performs the
     required
 *  bit modifications for the "Rate Select" bits of
     register
 *  0x0E.
 */
if(stopInterrupt){
    changeBits(false, 14, 2);
}
switch(freqOption){
    case 1:
        changeBits(false, 14, 3);
        changeBits(false, 14, 4);
        break;
    case 2:
        changeBits(true, 14, 3);
        changeBits(false, 14, 4);
        break;
        case 3:
            changeBits(false, 14, 3);
            changeBits(true, 14, 4);
            break;
        case 4:
            changeBits(true, 14, 3);
            changeBits(true, 14, 4);
            break;
        default:
            printf("Invalid Option\n");
            break;
    }
    writeToReg(CNTL, buf[14]);
    return;
}

void DS3231::clearAlarms(){

    /** \brief Clear the RTC Alarm Flags
     *
     *  Clears the Alarm 1 and Alarm 2 Flag bits of
         register 0x0F.
     */

    changeBits(false, 15, 0);
    changeBits(false, 15, 1);
    writeToReg(STAT, buf[15]);
    return;
}
```

A.6 main.cpp - Novel Function Test

```
#include "DS3231.h"

int main(){
    DS3231 test(true);
    test.readTimeAndDate();
    DS3231 test2;
    test2.readTimeAndDate();
}
```

A.7 test.cpp - Test Program

```
#include "DS3231.h"

int main(){
    char address = 0x00;
    DS3231 testObject(address);
    char times[] = {20, 30, 12};
    char alarms[] = {30, 30, 12, 2, 30, 12, 2};
    char date[] = {1, 2, 3, 20};
    testObject.clearAlarms();
    printf("Starting the DS3231 test application\n");
    printf("--------------------------------------------------\n");
    printf("Test 1: Read Time and Date:\n");
    testObject.readTimeAndDate();
    printf("--------------------------------------------------\n");
    printf("Test 2: Write Time (12:30:20):\n");
    testObject.writeTime(times);
    usleep(100000);
```

```
DS3231 testObject2(address);
testObject2.readTimeAndDate();
printf("——————————————————————————————————————\n");
printf("Test 3: Write Date (02/03/20):\n");
testObject.writeDate(date);
usleep(100000);
DS3231 testObject3(address);
testObject3.readTimeAndDate();
printf("——————————————————————————————————————\n");
printf("Test 4: Read Alarms:\n");
testObject.readAlarms();
printf("——————————————————————————————————————\n");
printf("Test 5: Write Alarms (2nd @ 12:30:30,\n 2nd @
    12:30):\n");
testObject.setAlarms(alarms);
usleep(100000);
DS3231 testObject4(address);
testObject4.readAlarms();
printf("——————————————————————————————————————\n");
printf("Test 6: Set Interrupt:\n");
testObject.setInterrupt(true, true);
printf("NOTICE: Please visually verify alarm is set!\
   n");
usleep(15000000);
printf("——————————————————————————————————————\n");
printf("Test 7: Square Wave Generator:\n");
testObject.sqWaveGen(1, true);
printf("NOTICE: Please visually verify Square Wave
   Generator is Functioning!\n");
usleep(100000);
```

```
    printf("--------------------------------------------\n");
    printf("Test 8: Read Temperature:\n");
    testObject.readTemp();
    printf("--------------------------------------------\n");
    printf("Test 9: Novel Function:\n");
    testObject.readTimeAndDate();
    testObject.sysTimeRTCInit();
    usleep(100000);
    DS3231 testObject5(address);
    testObject5.readTimeAndDate();
    printf("--------------------------------------------\n");
    printf("--------------------------------------------\n");
}
```

A.8 System Service Script

```
[Unit]
Description=RPI RTC Service
Before=getty.target

[Service]
Type=oneshot
ExecStartPre=/bin/sh -c "/bin/echo ds1307 0x68 > /sys/
    class/i2c-adapter/i2c-1/new_device"
ExecStart=/sbin/hwclock -s
RemainAfterExit=yes

[Install]
WantedBy=multi-user.target
```