DCU School of Electronic Engineering
Assignment Submission

Student Name(s):	**Alex Doherty**
Student Number(s):	**15508783**
Programme:	**M.Eng. in Electronic and Computer Engineering**
Project Title:	**EE513 Assignment 1**
Module code:	**EE513**
Lecturer:	**Derek Molloy**
Project Due Date:	**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 is a grave and serious offence 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, 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.

I have not copied or paraphrased an extract of any length from any source without identifying the source and using quotation marks as appropriate. Any images, audio recordings, video or other materials have likewise been originated and produced by me or are fully acknowledged and identified.

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 referencing guidelines found

at http://www.library.dcu.ie/citing&refguide08.pdf and/or recommended in the assignment guidelines.

I understand that I may be required to discuss with the module lecturer/s the contents of this submission.

I/me/my incorporates we/us/our in the case of group work, which is signed by all of us.

Signed: Alex Doherty

Assignment 1

Physical connection of the RTC module to the I2C bus.

The first task for this assignment was to connect the RTC module to the I2C bus. Figure 1 shows the physical connection which was connected following the diagram given in the assignment brief. Wires coming from the GND, VCC, SCL and SDA pins on the DS3231 connected to the Raspberry Pi using the I2C bus.

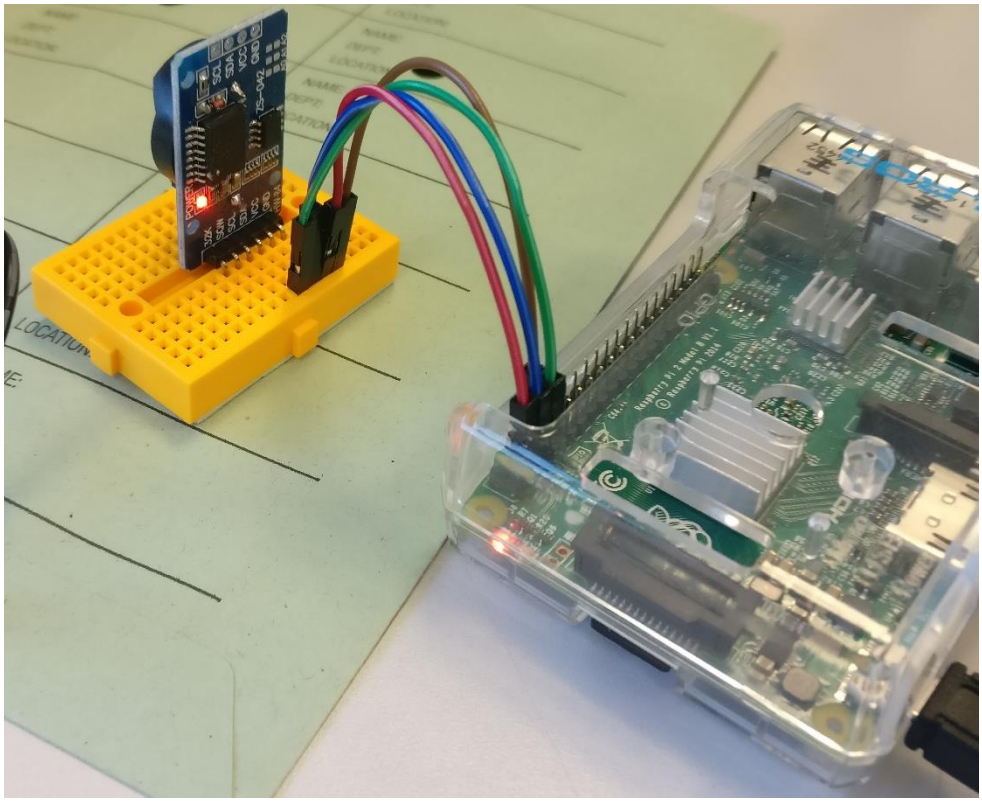


Figure 1:

Discovery and testing of the module using linux-based i2c-tools

The i2cdetect command is used to scan the I2C bus for devices. This command can be used to discover the device which is connected through the I2C bus.

```
pi@raspberrypi:~/Assignment1 $
pi@raspberrypi:~/Assignment1 $ i2cdetect -l
i2c-1   i2c             bcm2835 I2C adapter                     I2C adapter
pi@raspberrypi:~/Assignment1 $
```

Figure 2: i2cdetect command to scan I2C bus for devices

The module can be tested using the I2C tools, such as i2cset for setting registers that are visible through the I2C bus and i2cget which can be used to read registers that are visible through the I2C bus.

```
pi@raspberrypi:~ $ i2cset 1 0x68 0x02 0x04
WARNING! This program can confuse your I2C bus, cause data loss and worse!
I will write to device file /dev/i2c-1, chip address 0x68, data address
0x02, data 0x04, mode byte.
Continue? [Y/n] Y
pi@raspberrypi:~ $ i2cget 1 0x68 0x02
WARNING! This program can confuse your I2C bus, cause data loss and worse!
I will read from device file /dev/i2c-1, chip address 0x68, data address
0x02, using read byte data.
Continue? [Y/n] Y
0x04
pi@raspberrypi:~ $
```

Figure 3: i2cset and i2cget commands used to set and read a value in a register

In figure 3, the i2cset command writes the value 0x04 to the 8-bit register 0x02 of the I2C device at address 0x68 on bus 1 (i2c-1). The subsequent command reads the value from the register which was previously written to and prints out the value which is stored in that register. This is shown to be the value which was just written to the register 0x02 in figure 3.

Design, write and test a C++ class that wraps the functionality described in the DS3231 datasheet, including functionality to:

Read and display the current RTC module time and date

Figure 4 is a snippet of code which shows the C++ class reading in the registers stored at address 0x68. This function was based off the C code given in the assignment brief. [1]

```
int DS3231::readReg(int file)
{
    if(::read(this->file, this->buf, BUFFER_SIZE)!=BUFFER_SIZE){
        perror("Failed to read in the buffer\n");
        return 1;
    }
    return 0;
}
```

Figure 4: C++ code to read in registers

The function to read the time is shown in figure 5. There is a selection between two different functions, one which reads the time if it is a twelve hour clock and the other which reads the time if it is a twenty four hour clock. This choice is made when creating an object of the class, a value is set as 1 if it is a twelve hour clock or 0 if it is a 24 hour clock.

```
void DS3231::readTime(){
    if(twelve==1)
        readTwelve();
    else{
        readTwentyfour();
    }
}
```

Figure 5: C++ code to read and print the time of the RTC

The readTwelve function reads the time for a 12 hour clock. The registers are read in and then the appropriate registers are printed out to display the time. The lower 4 bits are read for the hours register as the upper bits will contain bits which determine if it's 12 or 24 hours, or AM/PM. The lower four bits are enough to store values for the 12 hour clock. The code to extract the four bits from the byte was based off of code online for this concept. [2] The bcdToDec function is provided in the C code in the assignment brief. [1] This is used to convert the value stored in the register to decimal before it is printed out.

```
void DS3231::readTwelve(){
    resetAdd(this->file);
    readReg(this->file);
    int hours, mask, y;
    hours = bcdToDec(buf[2]);
    mask = (1 << 4) - 1;
    y = hours & mask;
    printf("The RTC time is %02d:%02d:%02d\n", y, bcdToDec(buf[1]), bcdToDec(buf[0]));
}
```

Figure 6: C++ code for reading and printing the time for a 12 hour clock

The readTwentyfour function reads the time for a 24 hour clock. This function works similarly to the readTwelve function, except it reads 5 bits from the byte instead of 4. Five bits is enough to store 24, so that is all that is needed to be read.

```
void DS3231::readTwentyfour(){
    resetAdd(this->file);
    readReg(this->file);
    int hours, mask, y;
    hours = bcdToDec(buf[2]);
    mask = (1 << 5) - 1;
    y = hours & mask;
    printf("The RTC time is %02d:%02d:%02d\n", y, bcdToDec(buf[1]), bcdToDec(buf[0]));
}
```

Figure 7: C++ code for reading and printing the time for a 24 hour clock

The readDate function is used to read and print out the date, which are stored in registers 0x04, 0x05 and 0x06 for date, month and year respectively.

```
void DS3231::readDate(){
   resetAdd(this->file);
   readReg(this->file);
   printf("The RTC date is %02d/%02d/%02d\n", bcdToDec(buf[4]), bcdToDec(buf[5]), bcdToDec(buf[6]));
}
```

Figure 8: C++ code to read and print out the date

The code snippet in figure 9 shows test code in the main function, to ensure that the read function is working. An object of the class DS3231 is created and then the readTime function is called to output the time. The readDate function is also called to print out the date.

```
DS3231 i2c(1, 1, 1); //set first parameter to 1 if you want 12 hour clock, otherwise 0 for it to be 24 hours
                     //set second parameter to 0 for AM and 1 for PM
                     //Third parameter can be set to 1 = 1kHz, 2 = 1.024kHz,
                     //3 = 4.096kHz or 4 = 8.192kHz, for different frequency rates

printf("------------------------------------------------\n");
printf("------------Reading the date and time-----------\n");
printf("------------------------------------------------\n");
i2c.readTime();
i2c.readDate();
```

Figure 9: C++ code to test the task of reading the time and date

Figure 10 shows what is printed out when running the test code shown in figure 9. The RTC time is displayed along with the date.

```
------------------------------------------------
------------Reading the date and time-----------
------------------------------------------------
The RTC time is 12:23:14
The RTC date is 30/03/20
```

Figure 10: Test output to the terminal

Read and Display the Current Temperature

Figure 11 is a code snippet of the readTemp function which reads the registers and then prints the current temperature. The temperature is stored in two registers, 0x11 and 0x12. Unlike the time and date registers, the temperature is not stored in binary coded decimal, so the values are printed directly instead of passing them through any sort of function to convert them. The least significant byte of the temperature is shifted by six bits so that the data is stored in the lowest two bits. This can then give 4 results which are 00, 01, 10 and 11. The DS3231 datasheet [3] states that the temperature has a resolution of 0.25˚C. For this reason,

it is multiplied by 25, which will give a result of 0 for 00, 25 for 01, 50 for 10 and 75 for 11 for the fractional portion of the temperature.

```
void DS3231::readTemp(){
    resetAdd(this->file);
    readReg(this->file);
    printf("The temperature is %02d.%02d\n", buf[17], (buf[18]>>6)*25); //right bit shift LSB by
}                                                                        //6 bits to print relevant bits
```

Figure 11: C++ code to read and print the temperature

Figure 12 is the code used for testing the readTemp function, which is in the main function. Figure 13 shows an example of the output that will be displayed.

```
printf("-------------------------------------------------\n");
printf("-------------Reading the temperature-------------\n");
printf("-------------------------------------------------\n");
i2c.readTemp();
```

Figure 12: C++ test code for reading the temperature

```
-------------------------------------------------
-------------Reading the temperature-------------
-------------------------------------------------
The temperature is 21.25
```

Figure 13: Test code output to the terminal

Set the Current Time and Date on the RTC Module

The write time function is called to write time to the appropriate registers, to set the time on the RTC. The seconds and minutes are written to registers 0x00 and 0x01 respectively using the writeReg function. The writing of the hours is handled separately as some of the upper bits are set or cleared for different situations.

```
void DS3231::writeTime(){
    writeReg(0x00, decToBcd(0)); //write seconds
    writeReg(0x01, decToBcd(0)); //write minutes
    writeHours(0x02, 12); //write hours
}
```

Figure 14: C++ function for setting the RTC time

The writeReg function is used to write a value to a register. The register address and value are passed in as parameters to this function.

```
int DS3231::writeReg(unsigned int registerAddress, unsigned char value){
   unsigned char buffer[2];
   buffer[0]=registerAddress;
   buffer[1]=value;
   if (::write(this->file, buffer, 2)!=2){
      perror("I2C: Failed to write to the device\n");
      return 1;
   }
   return 0;
}
```

Figure 15: C++ code for writing to individual registers

The writeHours function contains two if statements. The first is if the twelve hour clock is intended on being used, the sixth bit of the value that will be stored in the register will be set. The second is if the clock will use AM or PM. The fifth bit of the value will be set if PM is being used. After this, the value will be written to the hours register.

```
void DS3231::writeHours(unsigned int registerAddress, char value){
      if(twelve == 1) //if object created is setting time in 12 hours
      {
         value |= (1 << 6);
         if(PM == 1) //if time to be set is pm
            value |= (1 << 5);
      }
   writeReg(registerAddress, decToBcd(value)); //write hours
}
```

Figure 16: C++ function for handling writing to the hours register

The writeDate function simply calls the writeReg function to write the values for the date, month and year into registers 0x04, 0x05 and 0x06 respectively.

```
void DS3231::writeDate(){
    writeReg(0x04, decToBcd(6)); //date
    writeReg(0x05, decToBcd(03)); //month
    writeReg(0x06, decToBcd(20)); //year
}
```

Figure 17: C++ function for setting the date

The values for the time and date can then be tested using the code in figure 18. The DS3231 object that was created is used to call the functions which will set the time and date. This code is contained within the main function for testing.

```
printf("-------------------------------------------------\n");
printf("----Setting and then reading the date and time----\n");
printf("-------------------------------------------------\n");
i2c.writeTime();
i2c.writeDate();
```

Figure 18: Test code for setting the date and time

An example of the output when reading the date and time which have just been set by the write functions is shown in figure 19.

```
-------------------------------------------------
----Setting and then reading the date and time----
-------------------------------------------------
The RTC time is 12:00:00
The RTC date is 06/03/20
-------------------------------------------------
```

Figure 19: Test code output to terminal

Set and Read the Two Alarms. Set an RTC Interrupt Signal Due to an Alarm Condition and Evaluate That It Works Correctly Using Physical Wiring.

The setAlarm1 function is used to set the first alarm. The first alarm can be set by writing from registers 0x07 to 0x0a.

```
void DS3231::setAlarm1(){
   writeAlarm(0x07, 33); //write seconds
   writeAlarm(0x08, 52); //write minutes
   writeAlarmHours(0x09, 10); //write hours
   writeAlarmDate(0x0a, 06); //set alarm date

   setAlarmIE(0x0e, 2);//set interrupt condition
   setAlarmIE(0x0e, 0);//set alarm1 interrupt enable
}

void DS3231::readAlarm1(){
   resetAdd(this->file);
   readReg(this->file);
   printf("The RTC alarm time is %02d:%02d:%02d on date: %02d\n", bcdToDec(buf[9]), bcdToDec(buf[8]),
                                                                  bcdToDec(buf[7]), bcdToDec(buf[10]));
}
```

Figure 20: C++ code for setting the first alarm and reading the time and date it's set to

The writeAlarm function clears bit 7 of the register which is passed in and then writes the value to the register. The reason for clearing bit 7 is that these are the mask bits and the alarm will only occur if these bits are 0.

```
void DS3231::writeAlarm(unsigned int registerAddress, char value){
   value &= ~(1 << 7); //clear bit
   writeReg(registerAddress, decToBcd(value));
}
```

Figure 21: Function for setting the alarm registers

There is a separate function for writing the hours of the alarm as, similarly to setting the time of the RTC, the upper bits of the alarm hours register contains bits which can be set or cleared depending on whether the time will be in twelve or twenty four hours and if the time will be AM or PM.

```
void DS3231::writeAlarmHours(unsigned int registerAddress, char value){
      if(twelve == 1) //if object created is setting time in 12 hours
         value |= (1 << 6);
      if(PM == 1) //if time to be set is pm
         value |= (1 << 5);
   writeReg(registerAddress, decToBcd(value)); //write hours
}
```

Figure 22: Function for handling setting the hours register for the alarm

For setting the date of the alarm, bits 6 and 7 are cleared. The seventh bit is the mask bit again and the sixth bit is whether you want the number set to represent the day or date. It is being cleared as this means it will represent the date. After this, the value is written to the appropriate address.

```
void DS3231::writeAlarmDate(unsigned int registerAddress, char value){
    value &= ~(1 << 7); //clear bit 7
    value &= ~(1 << 6); //clear bit 6
    writeReg(registerAddress, decToBcd(value)); //write hours
}
```

Figure 23: Function for setting the date for the alarm

The setAlarmIE function is used to set the alarm interrupt enable as well as the interrupt control (INTCN) bit. These need to be set for the alarm to activate the alarm condition and the square wave signal is output to the square wave pin when the INTCN bit is set to zero.

```
void DS3231::setAlarmIE(unsigned int registerAddress, int bit){
    buf[14] |= (1 << bit); //set bit
    writeReg(registerAddress, decToBcd(buf[14]));
}
```

Figure 24: Function for setting the interrupt enable for the alarm

The functions for setting and reading the second alarm are very similar to the first alarm. This alarm only has minutes, hours and date registers, so it doesn't specify as specific as seconds like in the first alarm. The registers for the second alarm are 0x0B to 0x0D.

```
void DS3231::setAlarm2(){
    writeAlarm(0x0b, 53); //write minutes
    writeAlarmHours(0x0c, 11); //write hours
    writeAlarmDate(0x0d, 05); //set alarm date

    setAlarmIE(0x0e, 2);//set interrupt condition
    setAlarmIE(0x0e, 1);//set alarm1 interrupt enable
}

void DS3231::readAlarm2(){
    resetAdd(this->file);
    readReg(this->file);
    printf("The RTC alarm time is %02d:%02d on date: %02d\n", bcdToDec(buf[12]), bcdToDec(buf[11]),
                                                              bcdToDec(buf[13]));
}
```

Figure 25: C++ code for setting the first alarm and reading the time and date it's set to

The alarm is evaluated using physical wiring by connecting an LED to the square wave pin. The LED will be on when the alarm condition has not been met and once it has been met, the LED will turn off.

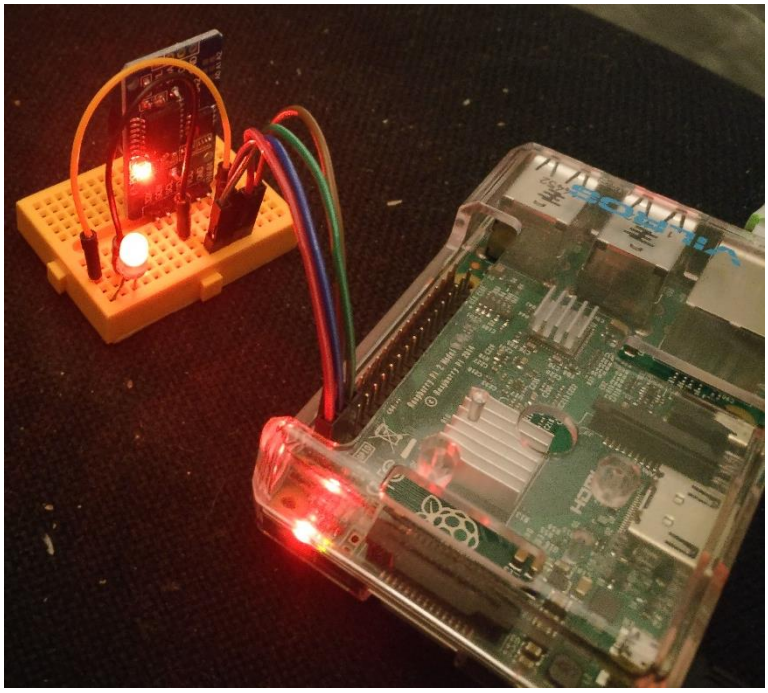


Figure 26: Wiring that connects the square wave output to the LED

Wrap the Square-Wave Generation Functionality of the RTC Module

A rateSelect function is used to select the frequency rate of the square wave. There are four different rates which are 1Hz, 1.024kHz, 4.096kHz and 8.192kHz. When creating objects of this class, a number must be set between 1 and 4 to select a frequency rate. Depending on the selected rate, different bits are set and cleared as shown in figure 27. The value will then be written to the appropriate register.

```
void DS3231::rateSelect(unsigned int registerAddress){
   if(RS==1){
      buf[14] &= ~(1 << 3); //clear bit 3
      buf[14] &= ~(1 << 4); //clear bit 4
      printf("RS1\n");
   }
      //setRS(0x0e, 0, 0);
   if(RS==2){
      buf[14] |= (1 << 3); //set bit 3
      buf[14] &= ~(1 << 4); //clear bit 4
      printf("RS2\n");
   }
   if(RS==3){
      buf[14] |= (1 << 4); //set bit 4
      buf[14] &= ~(1 << 3); //clear bit 3
      printf("RS3\n");
   }
   if(RS==4){
      buf[14] |= (1 << 3); //set bit 3
      buf[14] |= (1 << 4); //set bit 4
      printf("RS4\n");
   }
   writeReg(registerAddress, decToBcd(buf[14]));
}
```

Figure 27: Function for selecting the frequency rate for the square wave

Test code is created in the main function to call the rateSelect function and pass in the register which contains the rate select bits.

```
printf("-------------------------------------------------\n");
printf("------------Setting the frequency rate------------\n");
printf("-------------------------------------------------\n");
read.rateSelect(0x0e);
```

Figure 28: Test code for selecting the frequency rate

Add novel functionality of your own design to your class

The novel functionality designed for this class was a function which outputs the date, time and temperature to the DS3231_Output.txt file, which will be created in the current working directory upon running this function. The setfill and setw functions are used to pad the

numbers being written with zeroes if it is a single digit number. For example, this will write '09' to the file instead of '9'. A cast to int is used for the temperature as the buf variable is of type char and without passing it through a function such as bcdToDec or the bit shifting which both return an int, a char will not write the number correctly to the file. For this reason, it is being cast to an int.

```
void DS3231::fileWrite(){
   ofstream myfile;
   myfile.open ("DS3231_Output.txt");

   myfile << "-------------------------------------------------\n";
   myfile << "------Reading the date, time and temperature------\n";
   myfile << "-------------------------------------------------\n";
   int hours, mask, y;
   hours = bcdToDec(buf[2]) >> 0;
   mask = (1 << 4) - 1;
   y = hours & mask;
   myfile << "The RTC time is " << setfill('0') << setw(2) << y << ":" << setfill('0') << setw(2) << bcdToDec(buf[1])
          << ":" << setfill('0') << setw(2) << bcdToDec(buf[0]) << "\n";

   myfile << "The RTC date is " << setfill('0') << setw(2) << bcdToDec(buf[4]) << "/" << setfill('0') << setw(2)
          << bcdToDec(buf[5]) << "/" << bcdToDec(buf[6]) << "\n" ;

   myfile << "The temperature is " << (int)buf[17] << "." << setfill('0') << setw(2)
          << (buf[18]>>6)*25 << "\n"; //cast to int as buf is a char, bit shift seems to convert to int
   printf("%02d", buf[17]);

   myfile.close();
}
```

Figure 29: Function for writing the date, time and temperature to a file

Test code was created in the main function to ensure the date, time and temperature were being output to the DS3231_Output.txt file correctly. The function is called, and the file can then be checked on the filesystem once the program has executed.

```
printf("-------------------------------------------------\n");
printf("---------------Novel functionality---------------\n");
printf("-------------------------------------------------\n");
printf("Output date, time and temperature to DS3231_Output.txt\n");
read.fileWrite();
```

Figure 30: Test code for the file writing function

Use a Pre-Written Standard Linux RTC LKM for Integrating the RTC with the Linux OS

Commands that were given in the assignment brief [1] were followed to integrate the RTC with the Linux OS. These can be seen, along with their output in figures 31, 32 and 33.

```
pi@raspberrypi:~/Assignment1 $ ls /lib//modules/4.19.97-v7+/kernel/drivers/rtc/*1307*
/lib//modules/4.19.97-v7+/kernel/drivers/rtc/rtc-ds1307.ko
pi@raspberrypi:~/Assignment1 $ sudo modprobe rtc-ds1307
pi@raspberrypi:~/Assignment1 $ lsmod|grep rtc
rtc_ds1307             24576  0
hwmon                  16384  2 rtc_ds1307,raspberrypi_hwmon
pi@raspberrypi:~/Assignment1 $ sudo sh -c "echo ds1307 0x68 > /sys/class/i2c-adapter/i2c-1/new_device"
pi@raspberrypi:~/Assignment1 $ dmesg|tail -1
[  252.480045] i2c i2c-1: new_device: Instantiated device ds1307 at 0x68
pi@raspberrypi:~/Assignment1 $ ls -l /dev/rtc*
lrwxrwxrwx 1 root root      4 Mar  7 03:21 /dev/rtc -> rtc0
crw------- 1 root root 253, 0 Mar  7 03:21 /dev/rtc0
pi@raspberrypi:~/Assignment1 $
```

Figure 31: Associating the LKM with the bus device

```
pi@raspberrypi:~/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: -- -- -- -- -- -- -- -- UU -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Figure 32: i2cdetect now displays UU instead of 68 for the RTC device

```
pi@raspberrypi:~ $ date
Sat Jan  1 00:01:52 GMT 2000
pi@raspberrypi:~ $ sudo hwclock -r
2020-03-09 14:16:39.948019+00:00
pi@raspberrypi:~ $ sudo hwclock -w
pi@raspberrypi:~ $ sudo hwclock -r
2000-01-01 00:02:16.948119+00:00
pi@raspberrypi:~ $ sudo hwclock --set --date="2000-01-01 00:00:00"
pi@raspberrypi:~ $ sudo hwclock -r
2000-01-01 00:00:18.923393+00:00
pi@raspberrypi:~ $ sudo hwclock -s
pi@raspberrypi:~ $ date
Sat Jan  1 00:00:29 GMT 2000
```

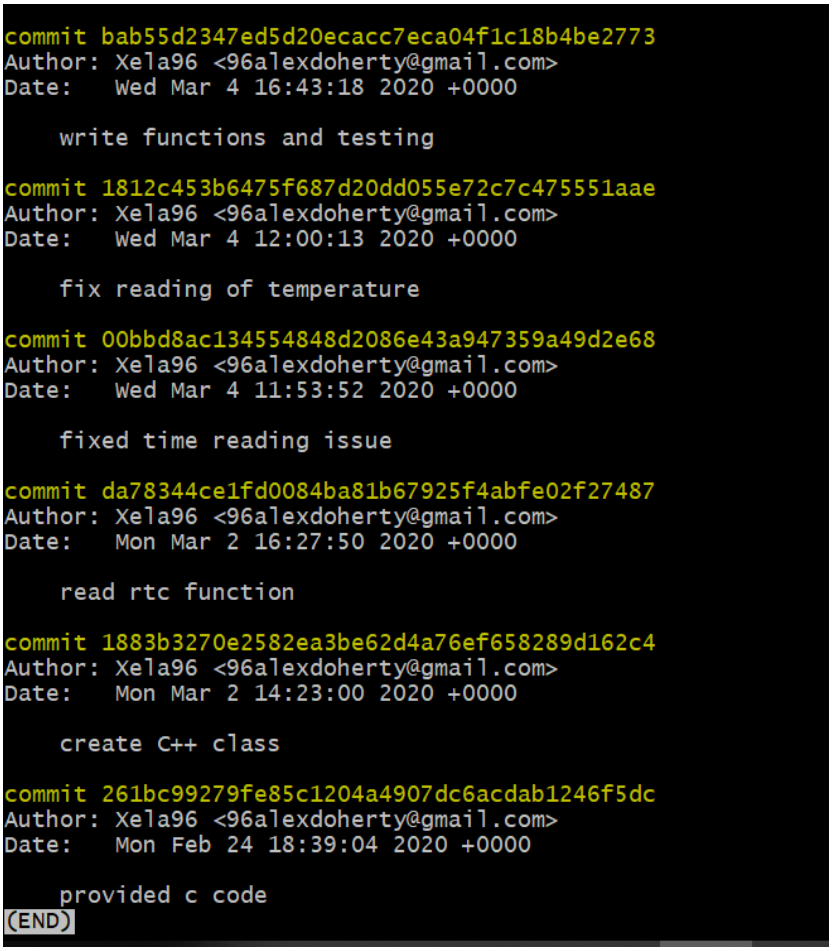
Figure 33: HWclock utility being used to read time from RTC clock, write time to the RTC clock and set the system clock

Conclusion

This assignment involved using the DS3231 RTC which is connected to the raspberry pi through an I2C bus. While all the individual tasks were completed for this assignment, a more clearly defined and structured approach should have been taken to the overall design of the C++ class. With more time, a separate class would be created for the I2C device. This class would contain the functions that interact with the I2C bus, such as opening the connection, reading and writing to the bus and closing the connection. There was also a lot of individual and very specific functions in this design of the class. This could be improved upon by making more

general functions, meaning the function could be used more often instead of designing a slightly different one for each scenario.

Commit History



```
commit 5a3812a0d55938066b720d38477f4653bcc2b8f1 (HEAD -> master)
Author: Xela96 <96alexdoherty@gmail.com>
Date:   Mon Mar 9 16:39:47 2020 +0000

    complete report

commit c55564f437f7cae2ce4fa21638d77e3dd9974623
Author: Xela96 <96alexdoherty@gmail.com>
Date:   Mon Mar 9 14:53:39 2020 +0000

    write report

commit c4ec8f7d04e2c06bb1caff45a1a4e4c4338c047e
Author: Xela96 <96alexdoherty@gmail.com>
Date:   Sun Mar 8 16:13:29 2020 +0000

    start report and clean code

commit 1faf82d2aadc8ea9a78b609d34cde5ee5532302e
Author: Xela96 <96alexdoherty@gmail.com>
Date:   Sun Mar 8 14:01:00 2020 +0000

    add novel functionality to class

commit dac6be5dba2415ce4d2977c51b2a6de67ffd1079
Author: Xela96 <96alexdoherty@gmail.com>
Date:   Sat Mar 7 17:29:38 2020 +0000

    square wave functionality

commit 6cf83fac2ab54ff9850c514b8c9880d35214a2a7
Author: Xela96 <96alexdoherty@gmail.com>
Date:   Fri Mar 6 17:44:19 2020 +0000

    handle upper nibble for writing time and set alarms

commit 243df5d4542898bc269a68bbdc181a034c57b5e5
Author: Xela96 <96alexdoherty@gmail.com>
Date:   Thu Mar 5 18:52:51 2020 +0000

    set upper bits for hours register

commit bab55d2347ed5d20ecacc7eca04f1c18b4be2773
Author: Xela96 <96alexdoherty@gmail.com>
Date:   Wed Mar 4 16:43:18 2020 +0000

    write functions and testing
```

References

[1] D. Molloy, *EE513 Assignment 1,* Dublin, 2020.

[2] Claudix, “How to read/write arbitrary bits in C/C++,” Stack Overflow, 5 August 2012. [Online]. Available: https://stackoverflow.com/questions/11815894/how-to-read-write-arbitrary-bits-in-c-c. [Accessed 6 March 2020].

[3] Maxim Integrated, “DS3231 Extremely Accurate I2C-Integrated RTC/TCXO/Crystal,” 2015. [Online]. Available: https://datasheets.maximintegrated.com/en/ds/DS3231.pdf. [Accessed 3 March 2020].