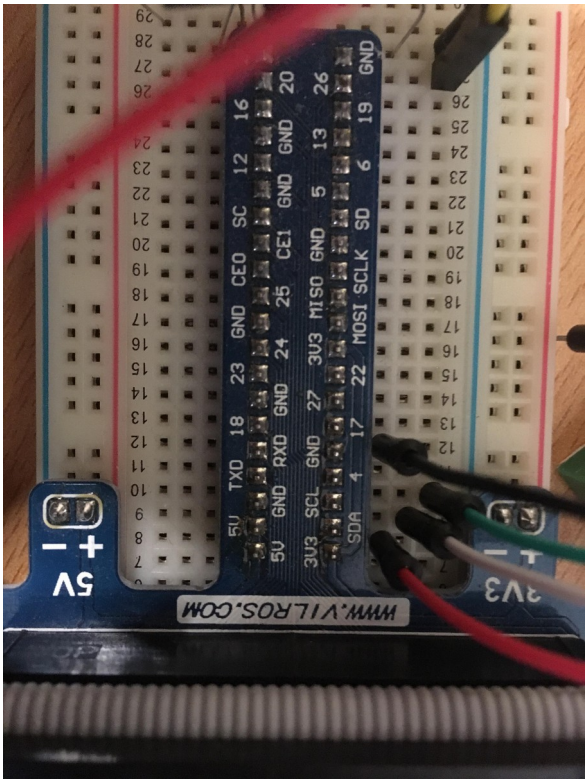
EE513 Assignment 1

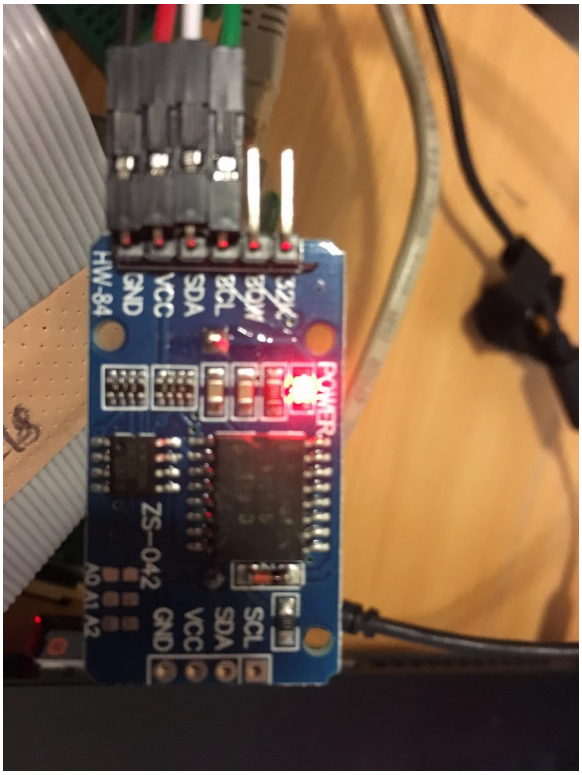
Liam Barry

Section 1 Introduction

1.1.Physical Connection of RTC Module to I2C Bus

The DS-3231 RTC clock was connected to a Raspberry Pi 2 Model B via a T-cobbler HAT for ease of identifying GPIO pins and protecting against accidental short circuiting. To enable I2C communication the SDA & SCL pins of the DS-3231 board were connected to the corresponding GPIO pins Fig 1.





1.2.Discovery and testing of the module using Linux-based i2c-tools

To verify correct wiring and become familiar with accessing the device registers the i2c-tools library (sudo apt-get install i2c-tools) was used. The “i2cdetect -l”command indicated that the DS-3231 was accessible under *dev*i2c-1. The default address of the DS3231 is 0x68 and this was confirmed using “i2c detect -y -r 1” whose output is displayed in Figure 2a. The registers can be directly read and printed to the terminal using the i2cdump command and the address of the DS3231. This was used extensively throughout the project to confirm successful manipulation of register data as in Figure below. Additional registers not corresponding to the DS3231 are omitted

```
pi@raspberrypi:~/assign/I2C_DEV/final_git $ sudo i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- 53 -- -- -- 57 -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- 68 -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

```
pi@raspberrypi:~/assign/I2C_DEV/final_git $ 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: 48 24 01 01 01 00 00 00 00 00 01 0a 94 7f 1f 89    H$???.....??????
10: 00 0e 00 XX XX XX XX XX XX XX XX XX XX XX XX XX    .?.XXXXXXXXXXXXX
```

Section 2

Design of a C++ Class to wrap DS3231 Functionality

Header and cpp files were created with the name DCLOCK which defined the DS3231 class and contained descriptions of member functions & states. This class inherited the public functionality of the supplied 'I2CDevice' class whose functions for opening a file handle, reading & writing registers etc were used extensively. A dedicated helper program named dtemp.cpp was used to create objects of this class and test their operation by printing to the output stream

2.1 Read & Display Time

displaytime()
The I2CDevice readRegister() function was used to assign three 'unsigned char' variables to the current state of the registers 0x00 to 0x02. In order to print the binary encoded decimal format to the output stream; static casting was implemented. Logic to handle the AM/PM and 12/24 hour formats was not completed but function produced accurate output when in 24 hour mode

```
void DCLOCK::displaytime(){

    unsigned char seconds = this->readRegister(0x00);
    unsigned char mins = this->readRegister(0x01);
    unsigned char hours = this->readRegister(0x02);
    cout << "The current time is" << static_cast<unsigned>(hours) <<
    ":" << static_cast<unsigned>(mins) << ":" << static_cast<unsigned>(seconds) << endl;
```

```
int main() {
    unsigned int bus = 1;
    unsigned int device = 0x68;
    DCLOCK dev2 = DCLOCK(bus, device);
    dev2.displaytime();
```

```
pi@raspberrypi:~/assign/I2C_DEV/final_git $ ./DCLOCK
The current time is: 00:00:01
```

displaydate()
Functionality almost identical to displaytime

2.2 Read and Display Temperature

show_temp()
The show_temp function worked similarly but required additional manipulation due to the different encoding of the register data. The temperature readings were stored in registers 11(MSB) & 12(LSB). The MSB used the entire 8 bits of register 11 while the LSB used just the upper 2 bits of register 12. Both bits were read in as ‘unsigned char’ and the lsb typecast to an integer to give the corresponding decimal value. Knowing that the additional 6 bits of the LSB register did not contain data and were read-only; the only 4 combinations were 00,01,10 and 11 in the first two bits followed by 6 trailing zeros. These corresponded to the decimal values 0,64,128 and 192 which were used to compare the data in the lsb register and set the output to .00, .25, .50 or .75 accordingly. The decimal value encoded by the MSB register was output using stringstream and typecasting

```
stringstream s;
s << (int)temp_msb;
string msb_s3 = s.str();

string lsb_s;
if(lsb == (int)0){
    lsb_s = ".00";}
else if(lsb == (int)64){
    lsb_s = ".25";
}
else if(lsb == (int)128){
    lsb_s = ".50";}
```

```
pi@raspberrypi:~/assign/I2C_DEV/final_git $ ./DCLOCK
Current Temperature is: 14.25C
```

2.3 Set Current Time & Date

set_time()
Writing to time registers was accomplished using arrays containing the address and value to be written in hexidecimal format. In the below example, the time is set to 19:32:35 with 0x13 being equivalent to the hour ‘19’ in decimal format. Logic to convert user decimal input into binary encoded time values was not completed and setting of time was hardwired as below.

```
void DCLOCK::set_time(){
    unsigned char hour_buf[] = {0x02,0x13};
    unsigned char min_buf[] = {0x01,0x20};
    unsigned char sec_buf[] = {0x00,0x23};

    this->writeRegister(hour_buf[0], hour_buf[1]);
    this->writeRegister(min_buf[0], min_buf[1]);
    this->writeRegister(sec_buf[0], sec_buf[1]);
```

set_date()
Similarly; writing to date registers was accomplished using arrays containing the address and value to be written in hexidecimal format. In the below example, the date is set to 09/03/20 with 0x14 being equivalent to the year 20 in decimal format

```
void DCLOCK::set_date(){
    unsigned char date_buf[] = {0x04,0x09};
    unsigned char month_buf[] = {0x05,0x03};
    unsigned char year_buf[] = {0x06,0x14};
    this->writeRegister(date_buf[0], date_buf[1]);
    this->writeRegister(month_buf[0], month_buf[1]);
    this->writeRegister(year_buf[0], year_buf[1]);
```

Output
The display functions explained above were used in conjunction with i2c tools to verify writing to the registers.

```
int main() {
    unsigned int bus = 1;
    unsigned int device = 0x68;
    DCLOCK dev2 = DCLOCK(bus, device);
    dev2.displaytime();
    dev2.set_time();
    dev2.displaytime();

    dev2.display_date();
    dev2.set_date();
    dev2.display_date();
```

```
pi@raspberrypi:~/assign/I2C_DEV/final_git $ ./DCLOCK
The current time is: 19:34:56
The current time is: 19:32:35
The date is: 9/3/20
```

2.4. Set and Read Both Alarms

Utilisation of the DS3231 alarms requires understanding of the control registers (0x0e & 0x0f) in addition to the alarm time registers (0x07-0x0d). First of all, the corresponding AE (Alarm Enable) bits in 0x0e must be set to high and the INTCN (interrupt control) bit must also be set to high if the square wave output pin is to be triggered in response to an alarm. Lastly; the corresponding Alarm Flag (AF) bits in 0x0f must be re-set to low (as far as the author could understand) in between alarms.

display_timealarm(int select_alarm)
Unlike the base time registers; the 8th bit (aka bit 7) of the alarm registers is used to set the alarm rate (see below). This prevents direct output of the register value when outputting the alarm time to the terminal (e.g. a value of 5 seconds would correspond to 0000-0101 in register 0x00 but potentially 1000-0101 in register 0x07). Bit-manipulation was used to clear the most significant bit of the values read-in from the registers **but these changes were not written to the device;** just the seconds variable within the function's scope.

```
void DCLOCK::display_timealarm(int select_alarm){
    unsigned char seconds = this->readRegister(0x07);
    seconds &= ~(1UL << 7); // for display
```

```
pi@raspberrypi:~/assign/I2C_DEV/final_git $ ./DCLOCK
The current time is: 00:02:35
In displaytime_alarm(): Alarm 1 is set to:  00:00:05
 With date /00
```

set_alrate(int setting, int alarm)
As per Table below there are 6 and 4 different settings for the alarm rate for Alarm 1 & 2 respectively. These can be adjusted by manipulating the MSB of the alarm registers and a function was created that accepted integer input of 1-6 corresponding to these settings as well as an additional input to select Alarm 1 or 2 and set the Mask Bits accordingly.

```
void DCLOCK::set_alrate(int setting, int alarm){
    if(alarm == 1){
    cout << "Setting Alarm 1 Mask Bits" << endl;
        if(setting == 1){
            //Alarm once per second
            bit_setter(0x07,7,1);
            bit_setter(0x08,7,1);
            bit_setter(0x09,7,1);
            bit_setter(0x0a,7,1);}
```

DY/$\overline{DT}$	ALARM 1 REGISTER MASK BITS (BIT 7)				ALARM RATE
	A1M4	A1M3	A1M2	A1M1	
X	1	1	1	1	Alarm once per second
X	1	1	1	0	Alarm when seconds match
X	1	1	0	0	Alarm when minutes and seconds match
X	1	0	0	0	Alarm when hours, minutes, and seconds match
0	0	0	0	0	Alarm when date, hours, minutes, and seconds match
1	0	0	0	0	Alarm when day, hours, minutes, and seconds match

DY/$\overline{DT}$	ALARM 2 REGISTER MASK BITS (BIT 7)			ALARM RATE
	A2M4	A2M3	A2M2	
X	1	1	1	Alarm once per minute (00 seconds of every minute)
X	1	1	0	Alarm when minutes match
X	1	0	0	Alarm when hours and minutes match
0	0	0	0	Alarm when date, hours, and minutes match
1	0	0	0	Alarm when day, hours, and minutes match

While writing of correct settings was confirmed; the different alarm settings proved hard to verify in testing. Once the alarm flag had been triggered the alarm would result in a single flash of the attached LED (see below) which did not repeat every second, when seconds matched etc. The datasheet indicated the alarm would enable the Square Wave output i.e. turn it on and keep it on; but this could not be demonstrated; only one off flashes in response to the alarm could be generated.

set_altime(int alarm)
Initial iterations of this function wrote hexidecimal values to the registers as with the set_time() function and are commented out of the functions source code. This had the potential to over-write the mask bits for the same reasons outlined above in the display_timealarm() function; and would require more bit handling or an additional call to the set_alrate() function to prevent over-writing. A less error prone method was to again use the bit_setter() to write values. Functionality to accept integer user input and convert to binary values for writing was not completed but as a proof of concept the alarm time was set to 00:00:05 and date to 01. The Day/Date bit was also set to Date mode (Low). It was assumed that setting the corresponding Alarm Flag low would always be desirable when setting the alarm time i.e. the function should take care of everything necessary to set up the alarm.

A helper function was also designed for testing that allowed the binary state of the A1F/A2F bits to be printed to the terminal to confirm alarm triggering. It was intended to poll this bit to further utilise the alarm functionality but this was not completed. Alarm triggeringwas physically confirmed using an LED connected via MOSFET to the Square Wave pin. The LED is illuminated by default but triggering of the alarm causes the Square Wave output to apply current to the gate of the MOSFET and switching off the LED. The code and output below show the alarm flag being re-set in the set_altime() function and confirming the time/date setting for the alarm as well as the current time in the terminal. Then, the time is reset using set_time(); the alarm flag bit is confirmed as being low, the thread sleeps for 6 seconds to allow the RTC time to match the alarm time and the switching of the flag bit is confirmed in the terminal. Accompanying photographs showing the turning off the LED in response to the alarm are in the appendix (Alarm Timelapse).

```
pi@raspberrypi:~/assign/I2C_DEV/final_git $ ./DCLOCK
Alarm Register Status : 137(10001001)
Alarm status E REG after write:   5(00000101)
Alarm status F REG after write: 136(10001000)
In displaytime_alarm(): Alarm 1 is set to:  00:00:05
 With date /01
Binary state of alarm ctrl register 0x0f : 136(10001000)
Binary state of alarm ctrl register 0x0e :   5(00000101)
The current time is: 00:00:16
The date is: 1/3/20
The current time is: 00:00:01
Alarm Register Status : 136(10001000)
Alarm Register Status : 137(10001001)
```

```
int main() {
    unsigned int bus = 1;
    unsigned int device = 0x68;
    DCLOCK dev2 = DCLOCK(bus, device);
    dev2.set_alrate(1,2);//Alarm 1 set to 'Seconds Match'
    dev2.al_check();
    dev2.set_altime(1);
    dev2.display_timealarm(1);
    dev2.displaytime();
    dev2.display_date();
    dev2.set_time();
    dev2.displaytime();
    dev2.al_check();
    std::this_thread::sleep_for (std::chrono::seconds(6));
    dev2.al_check();
```

Section 3.1. Square Wave and Novel Functionality

start_sqwv(int frq_set, int INTCN)
The DS3231 posesses a square wave generator that can be set to 4 different frequencies using the RS1 & RS2 control bits in the 0x0e register. The square wave generator can be forced on by setting the interrupt control bit (INTCN) in the same register to low or it can be placed under control of the alarm by setting INTCN to high.

This function allows the user to set both the frequency and the control mode. As a novel function; the square wave pin was connected to the Gate of a MOSFET connected to Ground via two 110 Ohm resistors (in series) and an LED; allowing the LED to be manually controlled by the start_sqwv function or serve as a physical flag when the alarm was triggered (see Figures in appendix).

Section 4.1. Linux Hardware RTV Device LKM's

Instructions were followed to implement a loadable kernel module to control the DS3231 and use it to set the system time on boot.

Step 1 & 2

Locate suitable driver. Note, kernel version had to be asserted first and substituted into provided code. Kernel version **4.19.57-v7+**

```
pi@raspberrypi:~/assign/I2C_DEV/final_git $ sudo modprobe rtc-d
s1307
pi@raspberrypi:~/assign/I2C_DEV/final_git $ lsmod | grep rtc
rtc_ds1307             24576  0
hwmon                  16384  2 rtc_ds1307,raspberrypi_hwmon
```

```
pi@raspberrypi:~ $ ls /lib/modules/4.19.57-v7+/kernel/drivers/r
tc/*1307*
/lib/modules/4.19.57-v7+/kernel/drivers/rtc/rtc-ds1307.ko
```

Step 3

The dmesg command was used to confirm the new device was instantiated. Despite the registration errors the device appeared to work normally and a call to i2c dump & i2cget confirmed the address was no longer available

```
pi@raspberrypi:~ $ dmesg |tail -l
[17858.312071] Under-voltage detected! (0x00050005)
[17864.552060] Voltage normalised (0x00000000)
[18598.795350] Under-voltage detected! (0x00050005)
[18602.955360] Voltage normalised (0x00000000)
[19122.957298] Under-voltage detected! (0x00050005)
[19133.357222] Voltage normalised (0x00000000)
[20276.409687] rtc-ds1307 1-0068: registered as rtc0
[20276.410229] i2c i2c-1: new_device: Instantiated device ds130
7 at 0x68
[20282.083467] i2c i2c-1: Failed to register i2c client ds1307
at 0x68 (-16)
[20892.524633] i2c i2c-1: Failed to register i2c client ds1307
at 0x68 (-16)
```

```
pi@raspberrypi:~/assign/I2C_DEV/final_git $ 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: -- -- -- 53 -- -- -- 57 -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- UU -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
pi@raspberrypi:~/assign/I2C_DEV/final_git $
```

Step 4

```
pi@raspberrypi:~ $ date
Wed 11 Mar 13:28:06 GMT 2020
pi@raspberrypi:~ $ sudo hwclock -r
2020-03-11 13:28:15.919017+00:00
pi@raspberrypi:~ $ sudo hwclock -r
2020-03-11 13:28:39.986202+00:00
pi@raspberrypi:~ $ sudo hwclock --set --date="2000-01-01 00:00:
00"
pi@raspberrypi:~ $ sudo hwclock -r
2000-01-01 00:00:04.092610+00:00
```

A slight problem was observed during this step in that part of the hwclock output appeared distorted. This also seemed to prevent the hwclock from setting the system time as sudo hwclock -s did not change the output of the ‘date’ command.

Step 5

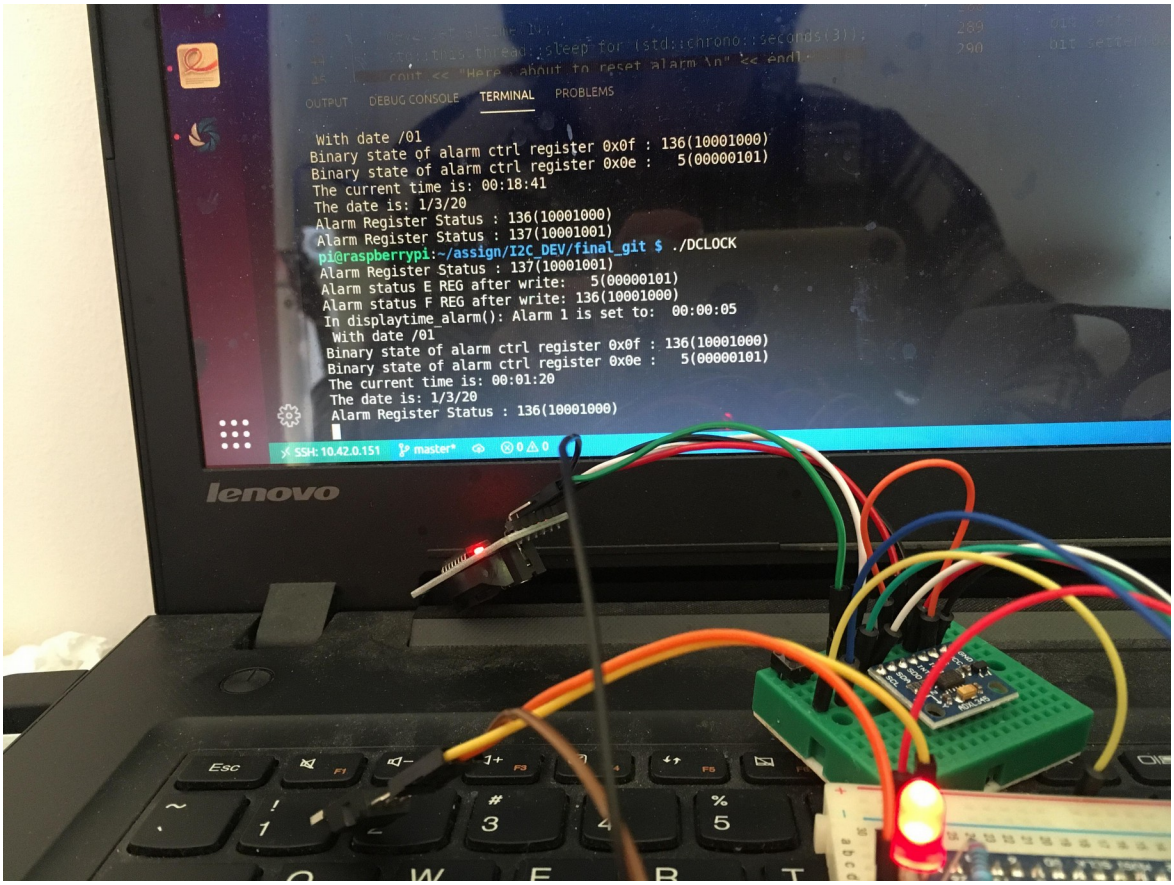
Nevertheless, the nano editor was used to create a new systemd file to attempt to set the system time from the RTC clock on boot and after correcting a small typing error in the provided systemd template the hwclock service was enabled.

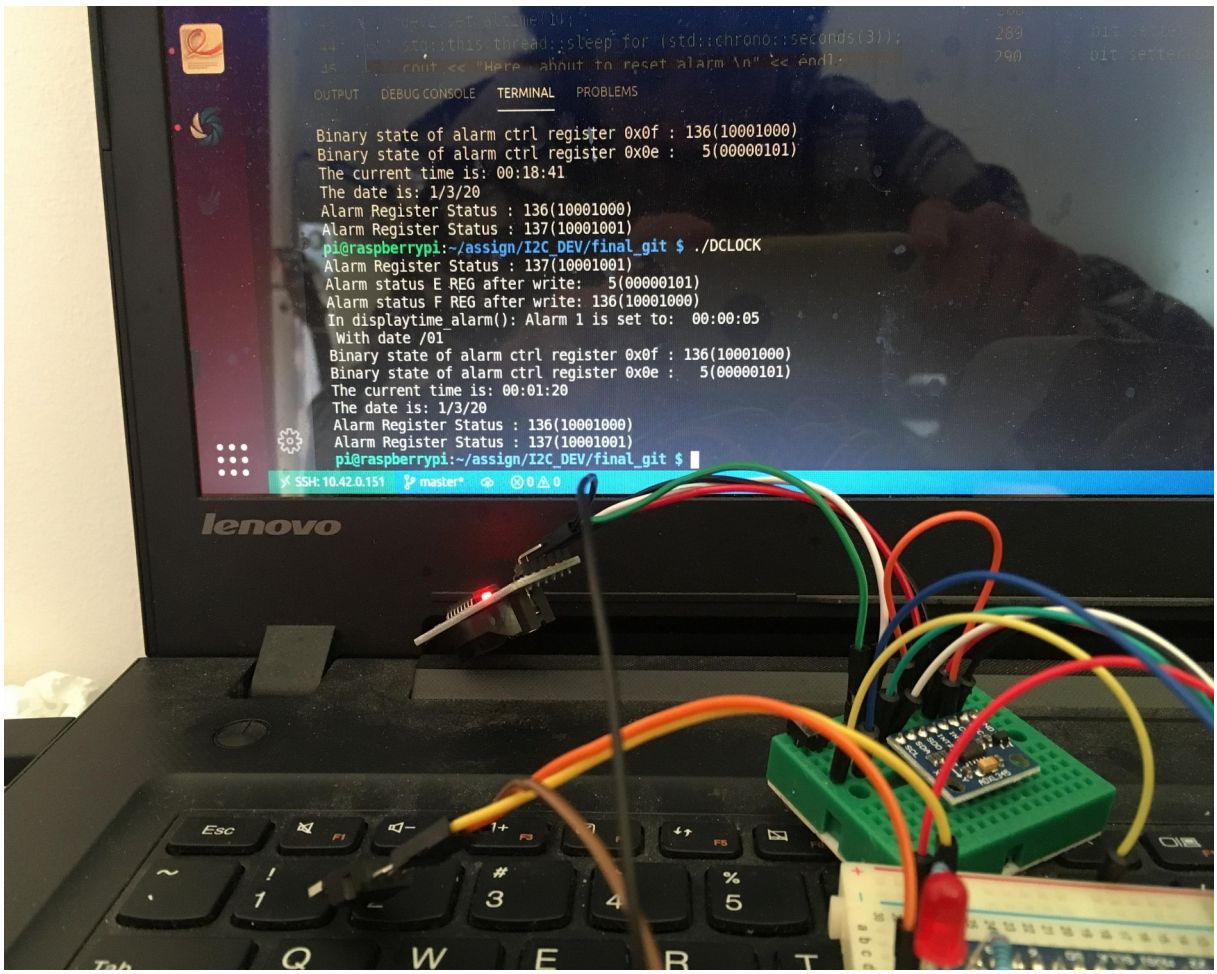
```
pi@raspberrypi:/lib/systemd/system $ sudo systemctl enable erpi
_hwclock
Failed to enable unit: File erpi_hwclock.service: Invalid argum
ent
pi@raspberrypi:/lib/systemd/system $ sudo nano erpi_hwclock.ser
vice
pi@raspberrypi:/lib/systemd/system $ sudo systemctl enable erpi
_hwclock
Created symlink /etc/systemd/system/multi-user.target.wants/erp
i_hwclock.service → /lib/systemd/system/erpi_hwclock.service.
```

Despite not being able to set the system clock with the RTC the module was successfully enabled at boot.

```
pi@raspberrypi:~/assign/I2C_DEV/final_git $ sudo systemctl status erpi_hwclock.service
● erpi_hwclock.service - ERPI RTC Service
   Loaded: loaded (/lib/systemd/system/erpi_hwclock.service; enabled; vendor preset: enabled)
   Active: active (exited) since Wed 2020-03-11 13:58:55 GMT; 1min 14s ago
  Process: 270 ExecStartPre=/bin/sh -c /bin/echo ds1307 0x68 > /sys/class/i2c-adapter/i2c-1/new_device (cod
  Process: 302 ExecStart=/sbin/hwclock -s (code=exited, status=0/SUCCESS)
 Main PID: 302 (code=exited, status=0/SUCCESS)

Mar 11 13:58:43 raspberrypi systemd[1]: Starting ERPI RTC Service...
Mar 11 13:58:55 raspberrypi systemd[1]: Started ERPI RTC Service.
lines 1-9/9 (END)
```





Alarm Timelapse: Note Alarm Flag Bit changing from 0 to 1 after Alarm Triggered