DS3231 Real Time Clock

1. About the Module

The DS3231 (figure 1) is a cheap but highly accurate RTC (Real Time Clock) module which communicates using I2C (Inter Integrated Circuit), which is a two-wire communication bus system. The IC includes a temperature-compensated crystal oscillator (TCXO). The module in figure 1 with the breakout pins is actually called ZS-042; the DS3231 is only the large IC at the top of the module, but both terms are used commonly.

A real time clock module such as the DS3231 holds a small battery cell which keeps the module running even when there is no power on the VCC pin. It keeps track of the time by using a small quartz oscillator which has a natural frequency of 32768 Hz ($2^{15}\mathrm{Hz}$). The frequency of the crystal must stay as close as possible to these 32.768kHz for the time to be accurately tracked. However, the frequency changes very slightly with temperature. Therefore, the module also contains a thermometer and evaluates the influence of the temperature on the frequency about once a minute. As the influence of a certain temperature is predictable, and the frequency can be changed by the capacitance of the system, it is possible to decently compensate for these small inaccuracies by adjusting the capacitance. This is done automatically by the module; therefore, it is a temperature-compensated crystal oscillator (TCXO).

In comparison to other cheap RTC modules, the DS3231 is highly accurate due to its temperature compensation resulting in about 2 min of drift maximum within one year of operation. Another huge advantage is the possibility of programming up to two different alarms. When these alarms are triggered, the module can pull the voltage on its SQW pin low triggering an interrupt in the MCU.

The module can be especially useful for saving energy during projects where energy efficiency is highly important such as in this garden pond monitoring station because it is supposed to be self-sustainable in the end. The MCU (ESP32 or Arduino) can enter a sleep mode where most components are switched off completely and the power consumption drops significantly. To wake the MCU up again, it is possible to attach an interrupt to one of the MCUs interrupt pins (GPIO 2 & 3 for Arduino UNO and all GPIOs for ESP32) to make a hardware interrupt. The interrupt can be for example a signal going from HIGH to LOW or the other way around. The DS3231 is capable of putting out a square wave / interrupt signal on the SQW pin at a certain point in time. It can thus be used for example to wake up the ESP32 from sleep mode once every hour. The ESP32 can then reset the alarm, take sensor measurements, transmit the data using MQTT and go back to sleep mode afterwards.

As the ESP32 will only take hourly measurements, most of the power consumption would occur while it is idle, waiting for another measurement to be taken. The combination of an RTC module and a hardware interrupt can thus save a lot of the battery’s energy in the final application.

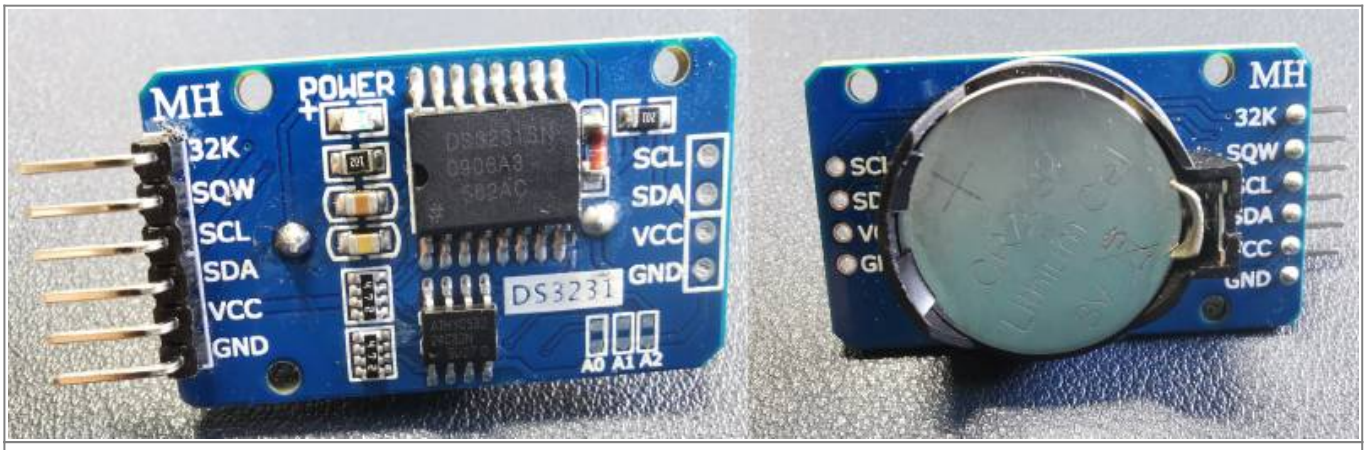


Figure 1 DS3231 RTC module

2 Data Transmission and Working Principle

2.1 I2C

As mentioned before, the DS3231 communicates using I2C. In an I2C bus, there is only one master device (the MCU) and up to 112 slave devices. In this case only the microcontroller and the RTC module are connected on the bus. The bus itself consists of a data line (SDA – Serial Data) and a clock signal (SCL – Serial Clock) issued by the master. Each device has its own I2C address which must be unique within the bus; for the DS3231 it is 104 by default but it can be changed by bridging the solder pads A0 to A2 (8 possible addresses).

Each communication starts with a start signal (start condition) by the master by pulling SDA LOW while SCL is HIGH and ends with a stop condition by pulling SDA HIGH while SCL is HIGH. The data transfer in between happens due to the master or the slave pulling the SDA signal low in certain intervals which can represent either a logic 0 or a logic 1 depending on the timing, similar to the 1-Wire bus used for the DS18B20.

After the start condition, the master sends the address (1 Byte) of the slave he wants to communicate with which only the slave with that address will react to. After sending the address, the master sends a single direction bit (R/W - Read/Write) representing the communication mode. If it is a logic 1, the master requests data to be sent from the slave (Slave Transmitter Mode). If it is a logic 0, the master sends data to the slave (Slave Receiver Mode). In either case the slave will send an acknowledge signal (ACK) by pulling SDA low. Afterwards the real data transmission starts.

In slave transmitter mode (figure 2), the slave will send one byte of data, followed by an ACK signal by the master. If the slave receives the ACK signal, it continues with the next byte and so on. The transmission ends when the master does not respond to a byte by sending a not acknowledge (NACK) signal which is followed by the stop signal.

In slave receiver mode (figure 3), the communication works the other way around, the master sends one byte, the salve sends an ACK signal and the next byte is transmitted. The communication stops when the master issues a stop signal after the slaves ACK signal.

The is also the possibility of the master sending a repeated start signal (SR) instead of a stop condition, after which another I2C address is sent and another transmission starts.

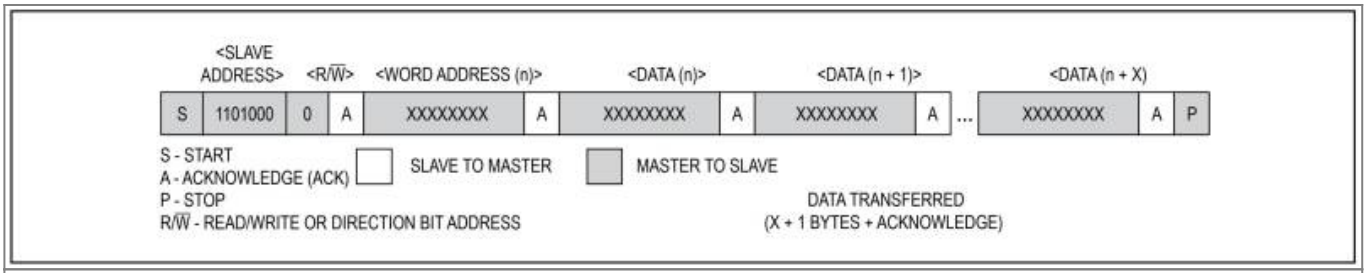


Figure 2 Slave Receiver Mode (Source: DS3231 Datasheet p.16)

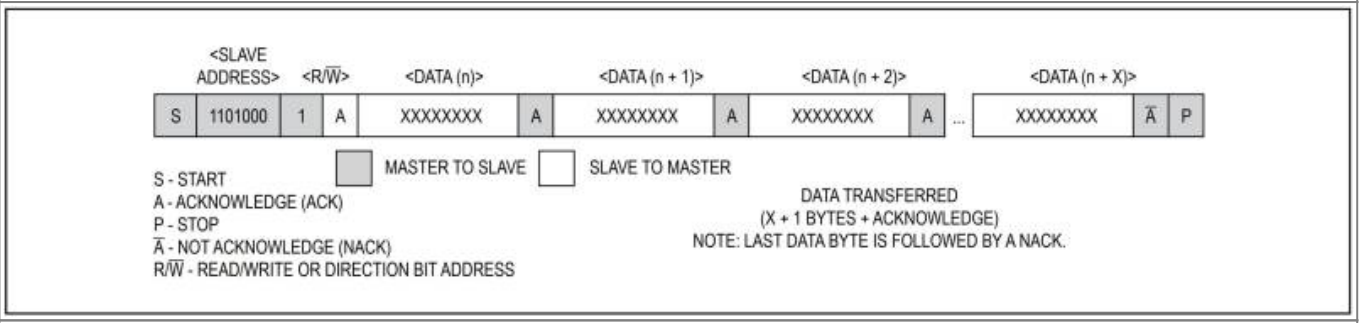


Figure 3 Slave Transmitter Mode (Source: DS3231 Datasheet p.16)

2.2 Registers

The module contains 19 8-bit registers (0x00 to 0x12) which can be used to read information from the module or change the modules settings (table 1). Registers 1 – 7 (0x00 – 0x06) contain the data for seconds, minutes, hours, day of the week, day of the month, month, and year, respectively. By writing into these registers, the time and date of the module can be configured. The module automatically updates these registers as time goes on, so that the actual time can be obtained by reading these registers later on. The module also automatically compensates for leap years and different lengths of the months.

Bit 6 of the hour register (0x02) can be used to change between 24 hour and AM/PM format where a 0 indicates 24 hour format. All of the registers (also the alarm registers) are written in binary coded decimal format. That means the bit 0 – 3 count the respective time units below a value of 10; so, they can have a value of 010 = B0000 to 910 = B1001. Depending on the register, bits 4 – 6 count the 10s of the time unit; the binary value they contain must be multiplied by 10 to obtain the decimal value. As an example, 57 minutes can be separated into 50 and 7. The 7 is written as 7 in bits 0 – 3 and 50 is written as 5 in the bits 4 – 6. The decimal (DEC) 57 written in binary coded decimal (BCD) would be B01010111. This is particularly important to consider when the alarm or time is set or read.

Register 8 – 11 (0x07 – 0x0A) contain the data for alarm 1. The registers store seconds, minutes, hours and date or day of the week. If the alarm is activated, the module checks every second if the time in the alarm matches the time in the time registers above. If that is the case, it can output an interrupt signal. The MSB (bit 7) gives information to the module which time data to consider. For example, can only check if the seconds of the registers match, or the seconds and minutes, or the seconds, minutes and hours and so on. Which value those bits (A1M1 – A1M4) need to have is listed in table 2. The 24 hour or AM/PM format can be selected the same way as in the time registers. Bit 6 in the day/date register determines if the days of the week or the date (day of the month) should be considered; a logic 1 selects day of the week, a logic 0 selects date.

The following registers 12 – 14 (0x0B – 0x0D) are the same as the previous ones but are for alarm 2. The only difference is that alarm two cannot check if the seconds match, it is thus not as accurately programmable. Register 15 (0x0E) is the control register which control different functions of the sensor. A logic 0 in the EOSC bit enables the oscillator to continue when the power supply is switched

to the battery. BBSQW is for choosing the output of the SQW pin of the module, if it contains a 1, it outputs a square wave, otherwise SQW can be used for sending the hardware interrupt to the MCU. Writing a logic 1 to the CONV bit (Conversion) issues a temperature measurement. RS2 and RS1 determine the frequency of the square wave and are not important in this project. A2IE enables the interrupt for alarm 2, which is not used here. A1IE (Alarm 1 Interrupt Enable) enables the interrupt for alarm 1 when logic 1 is written to it.

Register 16 (0x0F) is the status register containing information on the status of the module. OSF (Oscillator Stop Flag) should always be 0; if it contains a logic 1, there is something wrong with the oscillator. EN32kHz is for enabling the square wave signal. BSY (Busy) is set to logic 1 when the module is busy for example doing a temperature conversion. A2F and A1F (Alarm 1 Flag) contain a logic 1, when the alarm is triggered. The last three registers contain information on the capacitance for temperature compensation and on the last temperature measurement and are not important in this project.

In conclusion, the first registers are set to the desired time and are actualized continuously by the module. The following two groups of registers contain the alarm trigger time and the bit masks for which condition to check. The next two registers are only for enabling or disabling certain functions and reading operation data of the module.

An alarm is triggered when the alarm register matches with the time register which puts the alarm flag to a 1. If the alarm is enabled (A1IE) and interrupts are activated (INTCN), the SQW pin of the module pulls low which can then trigger an interrupt in the microcontroller and wake it up from sleep mode.

Table 1 Timekeeping Registers (Source: DS3231 Datasheet p.11)

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/$\overline{24}$	$\overline{AM}$/PM 20 Hour	10 Hour	Hour				Hours	1–12 + $\overline{AM}$/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
07h	A1M1	10 Seconds			Seconds				Alarm 1 Seconds	00–59
08h	A1M2	10 Minutes			Minutes				Alarm 1 Minutes	00–59
09h	A1M3	12/$\overline{24}$	$\overline{AM}$/PM 20 Hour	10 Hour	Hour				Alarm 1 Hours	1–12 + $\overline{AM}$/PM 00–23
0Ah	A1M4	DY/$\overline{DT}$	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/$\overline{24}$	$\overline{AM}$/PM 20 Hour	10 Hour	Hour				Alarm 2 Hours	1–12 + $\overline{AM}$/PM 00–23
0Dh	A2M4	DY/$\overline{DT}$	10 Date		Day				Alarm 2 Day	1–7
					Date				Alarm 2 Date	1–31
0Eh	$\overline{EOSC}$	BBSQW	CONV	RS2	RS1	INTCN	A2IE	A1IE	Control	—
0Fh	OSF	0	0	0	EN32kHz	BSY	A2F	A1F	Control/Status	—
10h	SIGN	DATA	DATA	DATA	DATA	DATA	DATA	DATA	Aging Offset	—
11h	SIGN	DATA	DATA	DATA	DATA	DATA	DATA	DATA	MSB of Temp	—
12h	DATA	DATA	0	0	0	0	0	0	LSB of Temp	—

Table 2 Alarm Mask Bits (Source: DS3231 Datasheet p.12)

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

3 Technical Specifications and Setup

As it was convenient to program, the DS3231 was programmed using the Arduino UNO instead of the ESP32. Later most of the code is not needed anymore as it only serves the initial setup of the RTC module. Here only the alarm for the DS3231 is programmed and the time is set. The handling of the interrupt signal is done in the implementation of the deep sleep mode of the ESP32 later on. For testing purposes only an LED was used to output a signal when the interrupt is triggered.

The module can be operated with 3.3 or 5V input voltage. It needs a CR2302 or similar sized 3V battery that can be plugged in on the backside so that operation continues when there is no power supply on the pins. The module has 6 pins, VCC and a GND pin which are connected to 3.3V and 0V, respectively. SDA is connected to A4 and SCL is connected to A5 on the Arduino. SQW is connected to GPIO 2. The 32kHz pin is not needed and left unconnected.

For testing, an LED was connected to GPIO 4 using a 330Ω resistor in series (figure 4). The operating temperature of the DS3231SN-IC is -40°C - 85°C but should be kept between 0°C – 70°C for more accurate results.

As the SQW is an open-drain output, it can only pull a HIGH signal to LOW. To work it needs an external pullup resistor. With the Arduino UNO and the ESP32, their internal pullup resistors can be used to achieve the same result.



There are different versions of the module, some of which have a red 1N4148 diode and a 200Ω resistor on them, which are connected to the battery. When power is supplied to the pins, this can be used to charge the battery. However, the battery is only a 3V battery and connecting the Arduino with 5V to VCC can overcharge the battery, reduce its lifetime significantly and pose a fire hazard. If the charging circuit is on the module, either 3.3V should be supplied or the circuit (diode or resistor) should be removed or the connection on the PCB has to be cut.

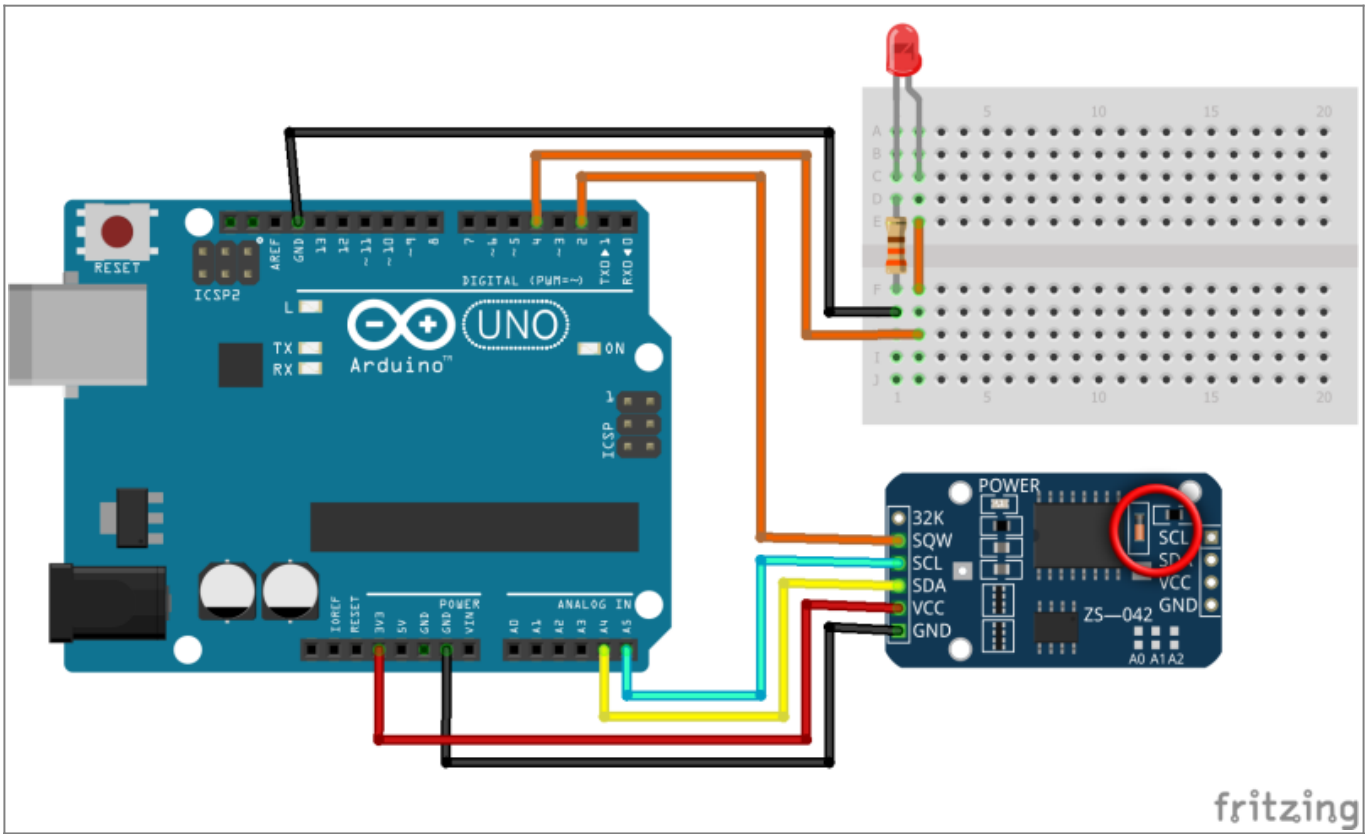


Figure 4 Schematic on how to connect Arduino, DS3231 and LED. The battery charging circuit is marked with a red circle.

4. Programming the DS3231

The aim for programming the DS3231 was to set the module's time registers correctly and set and activate the alarm to put out an interrupt signal once an hour to wake up the ESP32 from deep sleep in the final application.

There are a number of libraries available for the DS3231 RTC module, multiple of which were tested out. However, most of them are rather poorly documented, most have different features and some of them did not include the necessary function for the scope of the project.

In the end, after being inspired by the approach of Ralph Bacon, I programmed the DS3231 only using the `Wire.h` library for I2C communication without a dedicated DS3231 library.

The sketch by Ralph Bacon only included setting and reading the time from the module and printing the result to the serial monitor. Most of this was used with only minor changes in this sketch. The implementation of the interrupt and the setting and clearing of alarm 1 was done by me.

4.1 The Code

DS3231_Time_Alarm_set

```
//DS3231 Time read/set and Alarm

//Libraries
```

```
#include <Wire.h>                                   //1

//I2C Address of the Module
#define DS3231RTC_I2C_ADDRESS 0x68                  //2

//Day of the week
#define MONDAY 1                                    //3
#define TUESDAY 2
#define WEGNESDAY 3
#define THURSDAY 4
#define FRIDAY 5
#define SATURDAY 6
#define SUNDAY 7

//Alarm 1 Settings
#define ALARM_ONCE_PER_SECOND 0                     //4
#define ALARM_SECONDS_MATCH 1
#define ALARM_SECONDS_MINUTES_MATCH 2
#define ALARM_SECONDS_MINUTES_HOURS_MATCH 3
#define ALARM_SECONDS_MINUTES_HOURS_DATE_MATCH 4
#define ALARM_SECONDS_MINUTES_HOURS_DAY_MATCH 5
//Alarm 1 Mask Bits Array
byte Alarm1MaskBits [6]={B01111000,             //5
                         B01110000,
                         B01100000,
                         B01000000,
                         B00000000,
                         B10000000};

//Arduino UNO Interrupt pin and LED Pin
const uint8_t IntPin = 2;                       //6
const uint8_t LEDPin = 4;
bool ledstatus = 0;

//Interrupt Service Routine Variable
volatile byte Count = 0;                        //7

void setup(){
  Serial.begin(9600);

  Wire.begin();                                                 //8
  pinMode(IntPin, INPUT_PULLUP);                                //9
  pinMode(LEDPin, OUTPUT);
//10
  //Set time: Seconds, Minutes, Hours, Day, Date, Month, Year
  setRTCTime(0,15,10,FRIDAY,24,7,20);
//11
  //Set alarm 1: Seconds, Minutes, Hours, Day/Date, Setting
  setRTCAlarm1(0,30,12,12, ALARM_ONCE_PER_SECOND);
//12
  attachInterrupt(digitalPinToInterrupt(IntPin), ISRLED, FALLING);
```

```
//13
}

void loop(){
  clearAlarm1();                                  //14
  if(Count==1){                                   //15
    Count = 0;
    ledstatus = !ledstatus;
    digitalWrite(LEDPin, ledstatus);
  }
}

//Convert from decimal to binary coded decimal
byte decToBCD(byte val){                          //16
  return (((val/10)<<4)+val%10);
}

//Convert from binary coded decimal to decimal
byte bcdToDec(byte val){                          //17
  return (10*(val>>4) + val%16);
}

void ISRLED(){                                    //18
  Count++;
}

//Set the RTC Time Registers                      //19
void setRTCTime(byte Second, byte Minute, byte Hour, byte Day, byte
Date, byte Month, byte Year){
  Wire.beginTransmission(DS3231RTC_I2C_ADDRESS);//20
  Wire.write(0x00);                               //21
  Wire.write(decToBCD(Second));                   //22
  Wire.write(decToBCD(Minute));
  Wire.write(decToBCD(Hour));
  Wire.write(decToBCD(Day));
  Wire.write(decToBCD(Date));
  Wire.write(decToBCD(Month));
  Wire.write(decToBCD(Year));
  Wire.endTransmission();                         //23
}

//Set Alarm 1                                     //24
void setRTCAlarm1(byte Second, byte Minute, byte Hour, byte DayDate,
byte Setting){

  Second = decToBCD(Second) + bitRead(Alarm1MaskBits[Setting],3)*128;
//25
  Minute = decToBCD(Minute) + bitRead(Alarm1MaskBits[Setting],4)*128;
  Hour = decToBCD(Hour) + bitRead(Alarm1MaskBits[Setting],5)*128;
  DayDate = decToBCD(DayDate) + bitRead(Alarm1MaskBits[Setting],6)*128
```

```
+ bitRead(Alarm1MaskBits[Setting],7)*64;

  Wire.beginTransmission(DS3231RTC_I2C_ADDRESS);
  Wire.write(0x07);                                  //26
  Wire.write(Second);
  Wire.write(Minute);
  Wire.write(Hour);
  Wire.write(DayDate);
  Wire.endTransmission();
  Wire.beginTransmission(DS3231RTC_I2C_ADDRESS);
  Wire.write(0x0E);                                  //27
  Wire.write(B00011101);                             //28
  Wire.endTransmission();
}

//Clear Alarm 1
void clearAlarm1(){                                  //29
  Wire.beginTransmission(DS3231RTC_I2C_ADDRESS);
  Wire.write(0x0F);
  Wire.write(B00000000);
  Wire.endTransmission();
}

//Read the RTC Time Registers                        //30
void readRTCTime(byte* Second, byte* Minute, byte* Hour, byte* Day,
byte* Date, byte* Month, byte* Year){

  Wire.beginTransmission(DS3231RTC_I2C_ADDRESS);
  Wire.write(0x00);                                  //31
  Wire.endTransmission();

  Wire.requestFrom(DS3231RTC_I2C_ADDRESS, 7);        //32
  *Second = bcdToDec(Wire.read());                   //33
  *Minute = bcdToDec(Wire.read());
  *Hour = bcdToDec(Wire.read());
  *Day = bcdToDec(Wire.read());
  *Date = bcdToDec(Wire.read());
  *Month = bcdToDec(Wire.read());
  *Year = bcdToDec(Wire.read());
}

//Display the Time over the Serial Monitor
void displayTimeSerial(){                            //34
  byte Second, Minute, Hour, Day, Date, Month, Year;
  readRTCTime(&Second, &Minute, &Hour, &Day, &Date, &Month, &Year);

  Serial.print(Hour);
  Serial.print(":");
  if(Minute <10) Serial.print("0");
  Serial.print(Minute);
  Serial.print(":");
```

```
  if(Second <10) Serial.print("0");
  Serial.print(Second);
  Serial.print("   ");
  Serial.print(Date);
  Serial.print("/");
  Serial.print(Month);
  Serial.print("/");
  Serial.print(Year);
  Serial.print("   Day of the week: ");
  switch(Day){                                    //37
    case 1:
      Serial.println("Monday");
    break;
    case 2:
      Serial.println("Tuesday");
    break;
    case 3:
      Serial.println("Wednesday");
    break;
    case 4:
      Serial.println("Thursday");
    break;
    case 5:
      Serial.println("Friday");
    break;
    case 6:
      Serial.println("Saturday");
    break;
    case 7:
      Serial.println("Sunday");
    break;
  }
}
```

4.2 The Code Explained

1. only the **Wire.h** library for I2C