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.

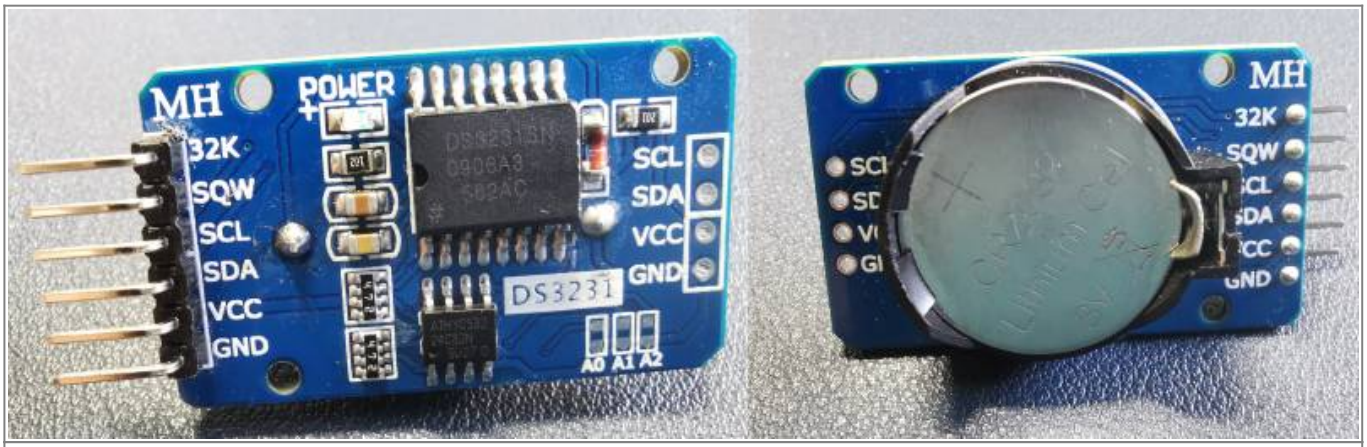

Figure 1 DS3231 RTC module

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.

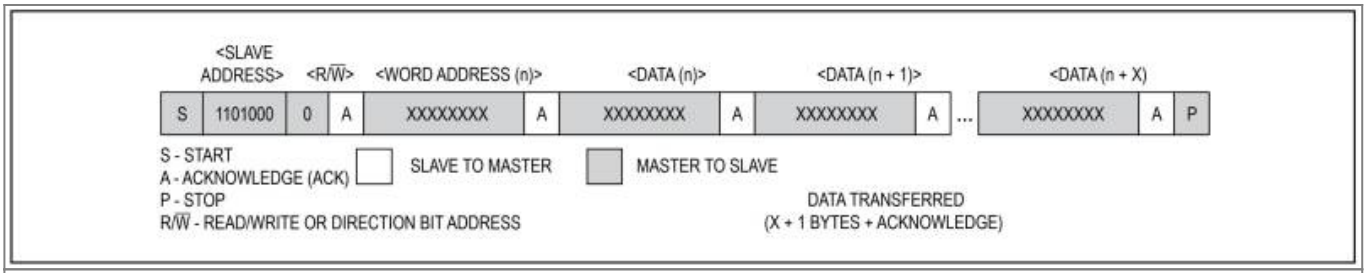

Figure 2 Slave Receiver Mode (Source: DS3231 Datasheet p.16)

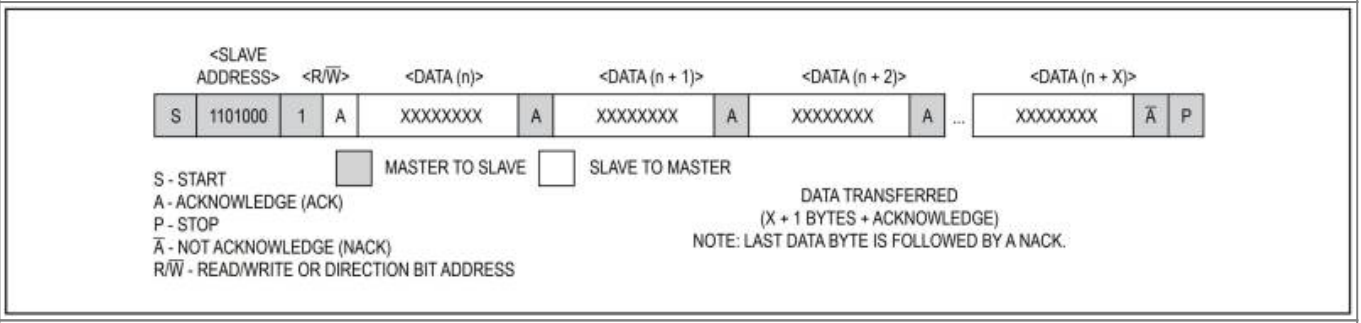

Figure 3 Slave Transmitter Mode (Source: DS3231 Datasheet p.16)

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.

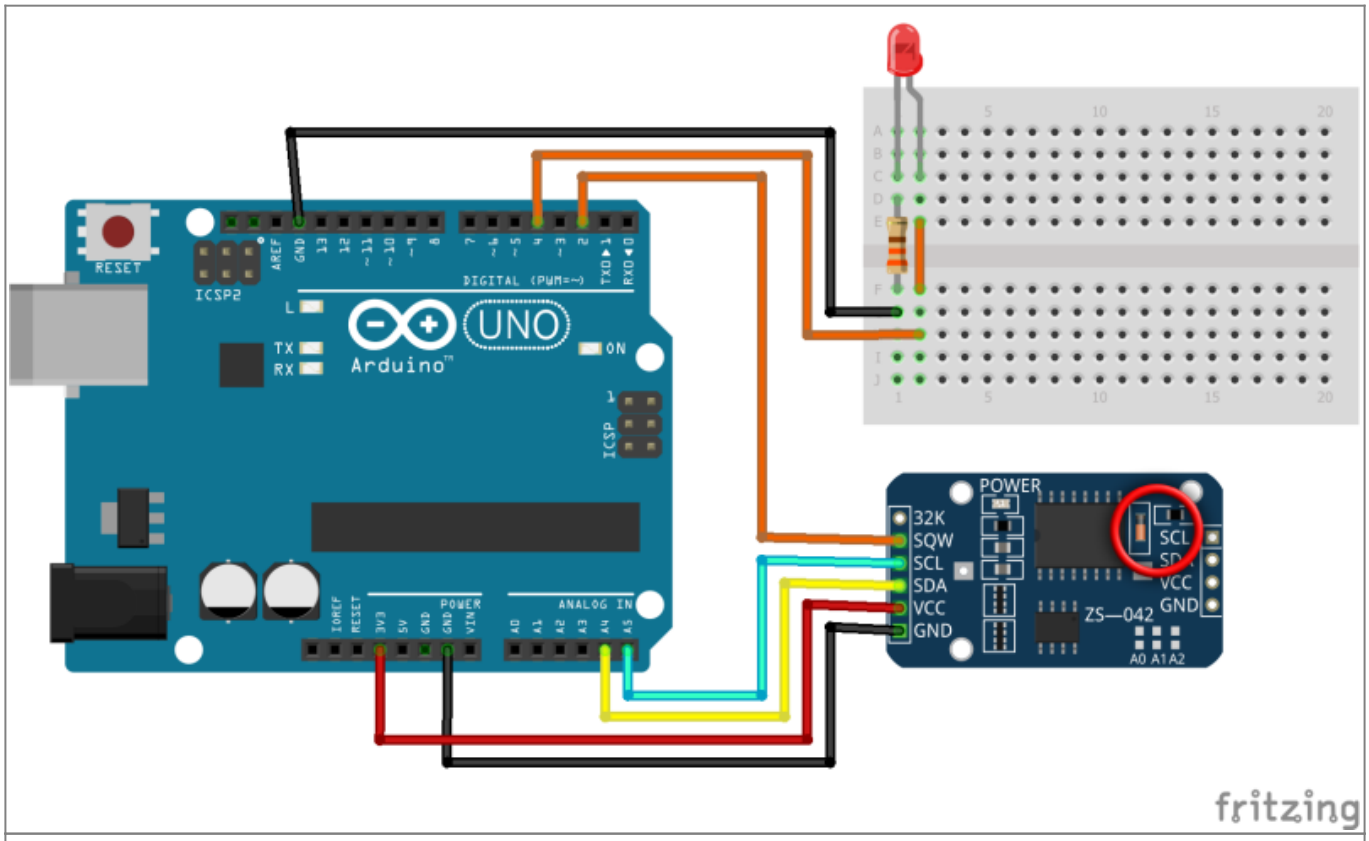

Figure 4 Schematic on how to connect Arduino, DS3231 and LED. The battery charging circuit is marked with a red circle.

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. As explained before, no external library needs to be included, only the **Wire.h** library for I2C communication for reading and writing to the registers of the DS3231.
2. The I2C address of the DS3231 needs to be defined. By default (if no solder connections are used) it is hexadecimal **0x68**.
3. When setting the day of the week later, 1 represents Monday, and 7 represents Sunday. To make it more intuitive the days are here defined.

4. As explained in section 2, the alarm register mask bits define the alarm rate. For example, choosing the **ALARM_SECONDS_MINUTES_MATCH** option would trigger an alarm once every hour - when the seconds and the minutes in the alarm and the time register match.
5. The **byte** array **Alarm1MaskBits[6]** is basically an array containing table 2 with the mask

bits for the alarm rates, where each array element represents a row from the table. The MSB in each element represents the bit for selecting either day or date format. Bit 6 is for A1M4, bit 5 is A1M3, and so on. The three LSBs are not needed and are just set to 0. Later on, using the previously defined (4) alarm 1 settings, one element is chosen from the array to set the alarm rate accordingly.

6. Here the Arduino pins for the interrupt (**IntPin**) and for the LED (**LEDPin**) are chosen. The interrupt pin has to be eitehr pin 2 or 3 for the UNO. the boolean variable **ledstatus** is just used in the main loop to switch the LED on and off.
7. When an interrupt is triggered, the MCU executes the attached interrupt service routine ISR which is defined later. The variable **Count** is used to count the number of interrupts triggered and is incremented by 1 in the ISR. Variables that are changed inside an ISR need to be configured as **volatile**.
   Configuring a variable as volatile tells the Compiler to load the variable from the RAM instead of the a storage register. This is necessary when the variable can be changed from somewhere else than the code that it is appearing in, for example a concurrently executed function like an ISR.

8. In the **setup()**, the method **Wire.begin()** starts the I2C bus connecting DS3231 and Arduino UNO.
9. The interrupt pin **IntPin** needs to be configured as an input. Furthermore, the DS3231's SQW pin (which is activated when an alarm is triggered) is an open drain output, which means it shorts the pin to GND when active. The Arduino interrupt pin (pin 2) needs to be HIGH normally, so that the voltage drop triggered by the SQW pin can be detected. As the Arduino has internal pullup resistors, activating them using **INPUT_PULLUP** is sufficient for this.
10. The pin for the LED needs to be configured as an **OUTPUT** to switch the LED on and off.
11. Here the later defined function **setRTCTime** is called to configure the time, date and day of the week of the DS3231. As an argument it needs the seconds, minutes, hours, day of the week, day of the month, month and year. After the time is configured for the first time, this should be commented out to prevent the time from being reset every time the Arduino restarts.
12. The function **setRTCAlarm1** configures the seconds, minutes, hours, date or day of the week, as well as the setting for the alarm rate for alarm 1 by writing to the alarm registers. Furthermore it activates alarm 1 and enables interrupts in the status register. The function is defined later on. If the alarm is already configured and working, this line of code can be commented out as well.
13. To attach an interrupt on the Arduino UNO, the function **attachInterrupt()** is used. As first argument the function **digitalPinToInterrupt()** is given and as its argument the Arduino interrupt pin (2) is submitted. Afterwards, the function expects the name of the interrupt service routine which is called whenever an interrupt is triggered; in this sketch it is called ISRLED. The last argument is the interrupt mode. There are 5 (for the UNO 4) different modes. Here the mode **FALLING** is necessary which indicates that the interrupt is triggered whenever the interrupt pin goes from HIGH to LOW.

14. The main **loop()** loop itself is very short. Firstly, by removing the alarm 1 flag in the DS3231 status register with the function **clearAlarm1()** defined below, alarm 1 is reset. In the final application this will only be done once after the MCU wakes up from deep sleep.
15. The **if** control structure checks the variable **Count**. In this example, the alarm was chosen to go off every second (12), so the **Count** variable is incremented by 1 once a second, so the condition becomes true after 1 second. It then resets the counter variable and changes the status of the LED. So in conclusion, it switches the LED on and off in 1 second intervals. The intervals can be changed by adjusting the condition. This is just used as an indicator to check if

the interrupt is working as supposed to.

16. The function **decToBCD()** returns and expects as an arguement a **byte** type value. It is used to convert a decimal number DEC to a binary coded decimal number BCD. As explained before, the time and alarm registers of the DS3231 are written in BCD format. However, DEC format is much more intuitive for the user to set the alarm and time of the module. Therefore, this function is used to convert the decimals given as function arguments for the two functions called in the setup (11 & 12) to the format that the DS3231 understands.
17. This function does the exact opposite of the previous one. It converts binary coded decimal back to decimal. This is only used for optionally checking if the time set in the DS3231 is correct.

18. The function **ISRLED()** is serving as the interrupt service routine called when an interrupt is triggered. Here it is used only to increment the counter variable and active the flag for the interrupt status. There are a few things to consider when writing an interrupt service routine. ISRs should always be as as short as possible because when they are called, the normal program stops as long as they are running; so functions that need a lot of time like print functions should be avoided. Furthermore,**delay()**, **millis()()** or **micros()** are not working inside an ISR and variables used in an ISR need to be **volatile**. More information can be found on the Arduino Reference page.

19. As explained before (11), **setRTCTime()** is used to configure the time registers of the DS3231.
20. The method **Wire.beginTransmission()** starts a transmission on the I2C bus in the slave receiver mode. As a function argument, the already defined I2C address (2) is given. Only the slave with that exact address will respond to the now following communication.
21. The method **Wire.write()** transmits the byte given as an argument. The first **write()** sets the register pointer of the DS3231. Here the pointer is set to **0x00**, which means that the following input is stored in the register with the address **0x00** which is the time register for the seconds.
22. Now the value for **Second** given as function argument is converted into BCD format and is then sent on the I2C bus to the DS3231 which overwrites the **0x00** register with the new information. That means the seconds of the DS3231 are configured. The register pointer then automatically jumps to the next register which is storing the minutes. As all the time registers are grouped together, all of them can be configured in one transmission, one after the other.
23. When all the registers are configured and the DS3231 has a new time set, the I2C transmission is stopped with **Wire.endTransmission()**.

24. Here follows the definition of function **setRTCAlarm1()** which was used in the **setup()** (12).
25. To configure the four registers of alarm 1, it is necessary to convert the seconds, minutes, hours and date/day into BCD format again. However, as can be seen in table 1, the alarm registers also contain the mask bits for the alarm rate which are stored as A1M1 to A1M4 in the MSBs of the four. The Which mask bit is used for which register is determined by the element chosen in the mask bit array (5).
    The function **bitRead()** is used to read a single bit from a byte. The first argument is the byte to read from; here the **Alarm1MaskBits[]** array is used and the byte element to read is determined by the **Setting** (4). **ALARM_ONCE_PER_SECOND** thus means the first element from the array which is **B01111000**, is chosen to read from. The second argument is the number of the bit, where 0 is the LSB and 7 is the MSB. The bit mask for the seconds register is the 4th bit from the right, so bit number 3. To now place that bit correctly in the **Second** byte, it mus be written inthe A1M1 bit (MSB). The MSB has a decimal value of 128, so the mask bit is just

multiplied by 128 which puts it to the front.
This same process is repeated for the minutes hours and days. What changes is the position of the respective mask bit in the mask bit array element; for minutes it is 4, for hours 5 and for days or date 6.
The hour format will automatically be 24 hours like in the time register because bit 6 stays always 0 (see table 1). However, when the alarm rate is chosen to be either once a week or once a month, i.e. days/date are matched, that has to be changed in the code. Therefore, the MSB of the array element contains a 0 for matching the date and a 1 for matching the day (table 2). This value is read from the array with another **bitRead()** and the retrieved value is multiplied with 64 to put it to bit 6 of the **DayDate** byte.

26. After configuring the values of the bytes for setting the alarm, another transmission to the RTC is started and the register pointer is set to register **0x07** containing the alarm settings for the seconds. Then the before calculated bytes are transmitted one after another to configure the alarm. and the transmission is ended.
27. To activate alarm 1 and enable the interrupt output throug the SQW pin, another transmission is started and the register pointer is set to **0x0E** which is the DS3231's control register.
28. Then the byte **B00011101** is transmitted through I2C to configure the control register. What it does can be found in detail in the datasheet on page 13. In short, what each bit does is: B: 0 - turn on the oscillator; 0 - no square wave output; 0 - no temperature conversion command; 1 - square wave frequency setting - 1 square wave frequency setting; 1 -activate interrupt output; 0 - deactivate alarm 2; 1 - activate alarm 1.
After that, the alarm is set and an interrupt is issued when the respective register entries match.

29. Whenever the alarm is activated, the alarm flag bit in the DS3231 status register is switched to a 1 and needs to be reset manually by changing the bit back to a 0. The function **clearAlarm1()** does exactly that. It starts a transmission, sets the register pointer to the status register and resets the value. More information can be found in the datasheet on page 14.

30. The last two functions are not used in the sketch but are useful to check whether the DS3231 has a correctly configured time register. The function **readRTCTime()** reads the values from the register and the function **displayTimeSerial()** prints it to the serial monitor. In the functions pointers are used to be able to only use local instead of global variables. The local variables **Second**, **Minute**, **Hour** and so on are declared locally in the **displayTimeSerial()** function. Then their addresses are given as function argument to the **readRTCTime()** which gives those addresses to its own locally declared pointer variables of the same name. The pointer variables are indicated by the asterisk * next to the variable name.
31. Then the I2C transmission is started in slave receiver mode to set the register pointer to the seconds register and the transmission is ended again.
32. Another transmission is started but in the slave transmitter mode using the **Wire.requestFrom()** method. It needs the I2C address of the slave and the number of bytes which are requested.
33. The dereferencing operator, the asterisk *, is used to change the value at the address of the pointer variable, in this case the value for the local variables from the **displayTimeSerial()** function. As a value they get the time stored in the DS3231 time registers converted back to decimal.

34. In the **displayTimeSerial()** function, the local variables are created, their addresses are given to the **readRTCTime()** function using the referencing operator & and then their results are printed to the serial monitor

4.3 Results

When uploading the sketch to the Arduino after connecting it according to the setup, the time of the module is set to Friday, 24th of July 2020, 10:15:00 which can be checked by calling the **`displaySerialTime()`** function. The alarm register is set to 12:30:00 and day/date 12 is chosen. Here it only makes sense if the alarm is not configured to match the days because a week does not have 12 days.

In the example the setting **`ALARM_ONCE_PER_SECOND`** is chosen which means that non of the alarm registers are matched to the time registers, it just gives a signal every second which can be seen by the LED switching on and off in 1 second intervals.

The 1 second intervals are of course only chosen to check whether the alarm works properly. As said, the ESP32 is to wake up once an hour, take measurements, transmit the data and go back to sleep afterwards. Therefore, for the final application, the setting **`ALARM_SECONDS_MINUTES_MATCH`** will be chosen. The minutes and seconds from the alarm register match exactly once per hour, so the alarm rate is correct.

Furthermore, the time in the alarm register will be set to 0 minutes and 0 seconds, such that the interrupt occurs with the beginning of every new hour, if the time registers are configured correctly.

After programming the alarm and time registers, most of the code will not be needed anymore and can be left out of the final sketch. The ESP32 will only need the I2C library, an interrupt service routine to wake it up, and the **`clearAlarm1()`** function to reset the alarm flag.

Back to the top