Interface DS3231 Precision RTC Module with Arduino

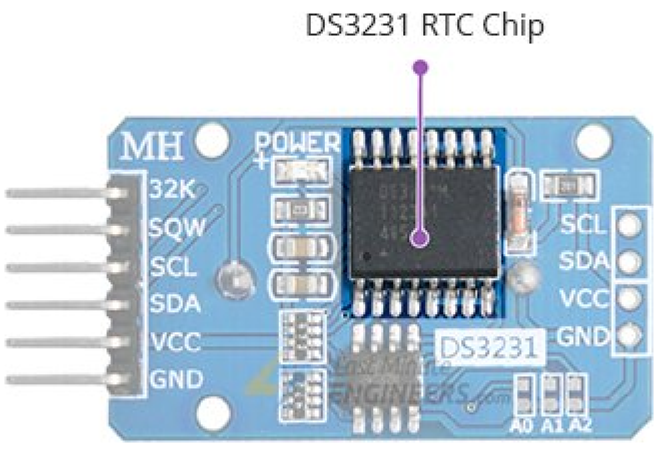
We are all aware that most MCUs used in our projects are time-agnostic; that is, they are unaware of the time around them. It's fine for most of our projects, but every now and then, when you come across an idea where keeping time is critical, the DS3231 Precision RTC module comes in handy. It is suitable for a wide range of projects, including clocks, timers, and alarms, as well as data logging and time stamping.

Hardware Overview

This module is built around the highly capable DS3231S RTC chip and the AT24C32 EEPROM, both of which have been around for a while and have good library support.

DS3231 RTC Chip

At the heart of the module is a low-cost, extremely accurate RTC chip from Maxim — the DS3231. It handles all timekeeping functions and communicates with the microcontroller over I2C.



The DS3231 can keep track of seconds, minutes, hours, days, dates, months, and years. For months with fewer than 31 days, it automatically adjusts the date at the end of the month, including leap year corrections (valid up to 2100).

It can work in either a 12-hour or 24-hour format and has an AM/PM indicator. It also has two time-of-day alarms that can be programmed.

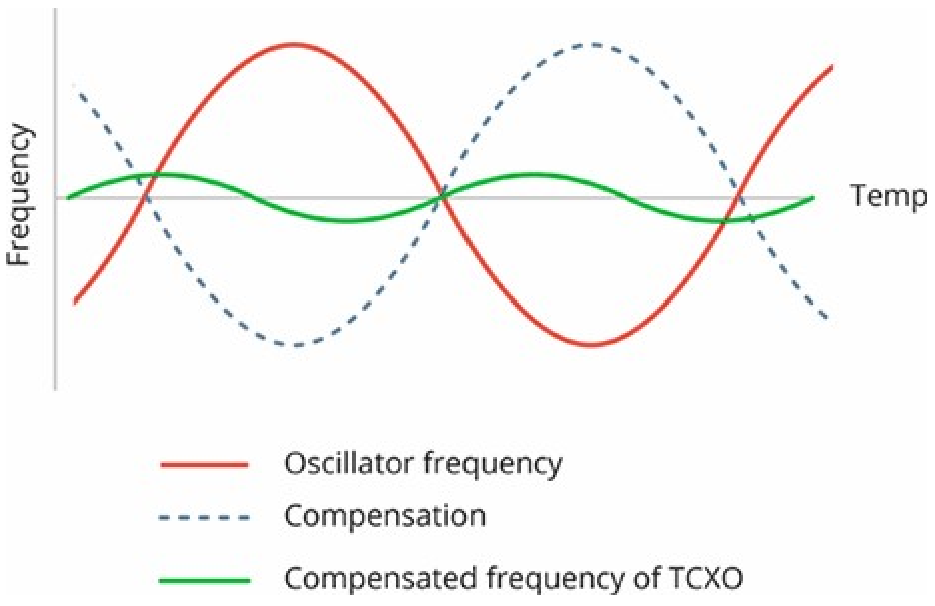
The INT/SQW pin on the DS3231 provides either an interrupt signal (due to alarm conditions) or a nice square wave at 1Hz, 4kHz, 8kHz, or 32kHz.

Additionally, the DS3231 outputs a stable (temperature compensated) and accurate reference clock on the 32K pin. This clock output may be useful in some applications for measuring clock timing accuracy or clocking other circuits.

Temperature Compensated Crystal Oscillator(TCXO)

Most RTC modules, such as the DS1307, require an external 32kHz crystal oscillator for timekeeping. The issue with these crystals is that their oscillation frequency is affected by external temperature. This change in frequency is negligible, but it adds up.

To avoid such minor crystal drifts, the DS3231 is powered by a 32kHz temperature compensated crystal oscillator (TCXO), which is highly resistant to external temperature changes.



The TCXO is actually composed of an integrated temperature sensor, a 32kHz crystal oscillator, and control logic. The integrated temperature sensor compensates for frequency changes by adding or removing clock ticks, ensuring that timekeeping remains accurate.

That is why the TCXO provides the most stable and accurate reference clock and keeps the RTC accurate to within ±2 minutes per year.

DS3231 Vs DS1307

The primary difference between the DS3231 and the DS1370 is the accuracy of timekeeping.

The DS1307 requires an external 32kHz crystal for timekeeping, the frequency of which is easily affected by external temperature. As a result, the clock is usually off by about five minutes per month.

The DS3231, on the other hand, is much more accurate because it has an internal Temperature Compensated Crystal Oscillator (TCXO) that isn't affected by temperature, making it accurate to a few minutes per year at most.

This doesn't mean that the DS1307 isn't accurate; it's still a great RTC that will serve you well, but if your project needs more accurate timekeeping, the DS3231 is a better choice.

Battery Backup

The DS3231 IC incorporates a battery input for maintaining accurate timekeeping even when the device's main power is interrupted.

The built-in power-sense circuit continuously monitors the status of VCC to detect power failures and automatically switches to the backup supply. This means that even in the event of a power outage, the IC can still keep time.

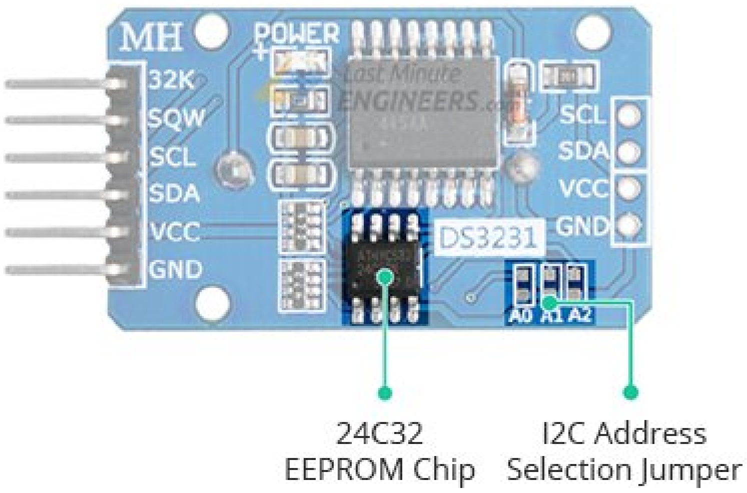
For backup power, the bottom of the board houses a battery holder for a 20mm 3V lithium coin cell.

Assuming you use a fully charged 220mAh coin cell battery and keep the chip current draw to a minimum of 3µA, the battery should be able to power the RTC for at least 8 years without the need for an external power supply.

220mAh/3µA = 73333.34 hours = 3055.56 days = 8.37 years

Onboard 24C32 EEPROM

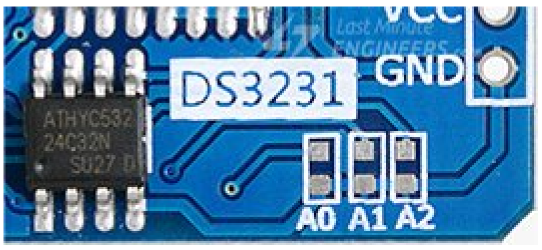
The DS3231 RTC module also includes a 32-byte (4K x 8-bits) AT24C32 EEPROM (non-volatile) chip with 1,000,000 write cycles. This chip doesn't really have anything to do with the RTC, but it can be useful for things like data logging or storing any other data that you want to be non-volatile.



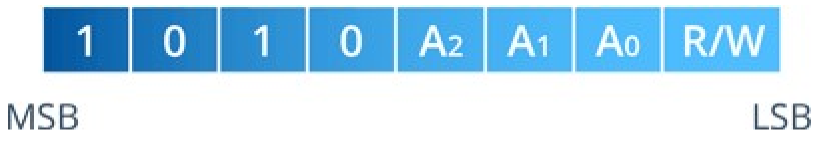
The 24C32 EEPROM communicates via I2C and shares the same I2C bus as the DS3231.

If you are using multiple devices on the same I2C bus, you may need to set a different I2C address for EEPROM so that it does not conflict with another I2C device.

To accomplish this, the module has three solder jumpers (A0, A1 and A2) on the back. Shorting a jumper with a blob of solder sets the address.



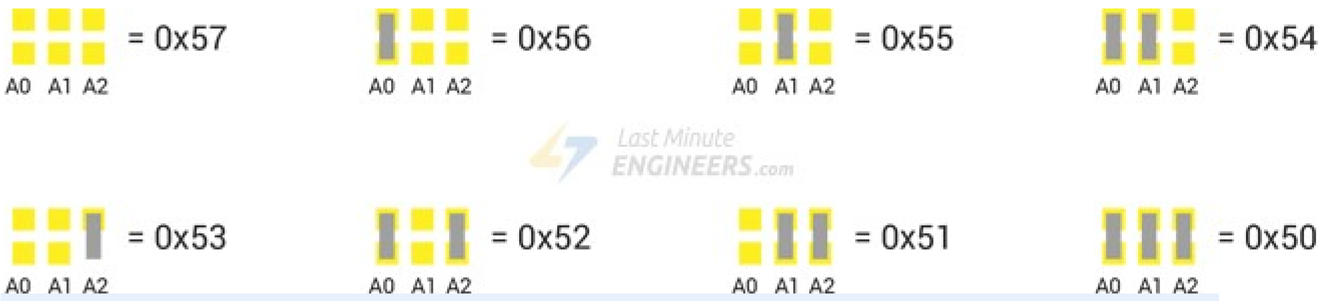
According to the datasheet for the 24C32, these three bits are placed at the end of the seven-bit I2C address, just before the Read/Write bit.



Because there are three address inputs that can be either HIGH or LOW, we can create eight (2^3) different addresses.

By default, all three address inputs are pulled HIGH using onboard pullups, giving the 24C32 a default I2C address of 1010111_{Binary} or $0x57_{Hex}$.

By shorting out the solder jumpers, the address inputs are pulled LOW. It allows you to set the I2C address according to the table below.

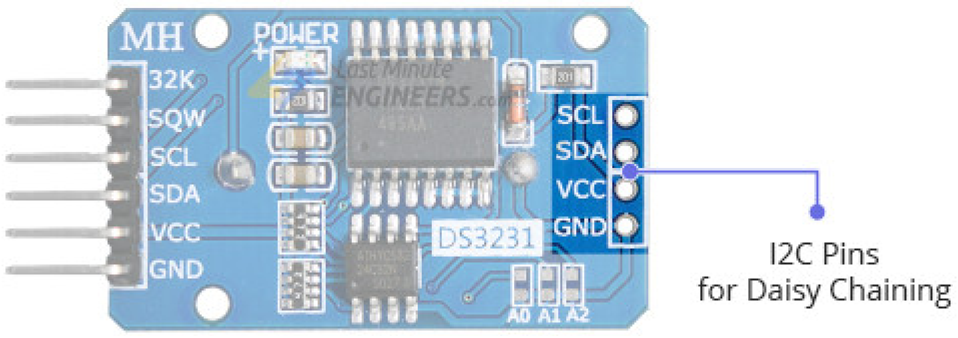


The AT24C32 EEPROM is rated for 1,000,000 write cycles, so it won't wear out during normal data logging applications as long as you're not writing data every second.

I2C Interface

The module has a simple I2C interface that takes up two addresses. The DS3231S RTC chip's fixed I2C address is 0x68, and the EEPROM's default I2C address is 0x57 (though the address range is 0x50 to 0x57).

The I2C SDA and SCL signals, as well as power and ground, are broken out to one side of the module to allow these signals to be looped out to another module.



To enable communication, both the SDA and SCL lines have 4.7K pull-up resistors.

Technical Specifications

Here are the specifications:

Operating Voltage	2.3 to 5.5V (3.3 or 5V typical)
Current Consumption	< 300µA (typ.)
Accuracy (0-40°C)	± 2ppm
Battery	CR2032 (3V Coin)

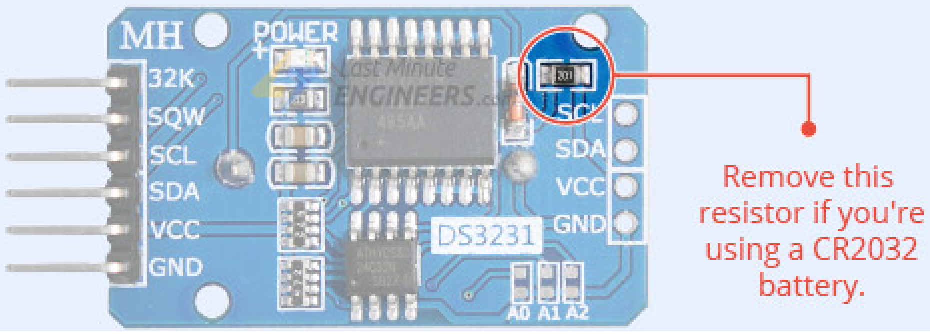
For more information on the DS3231 RTC and the 24C32 EEPROM chip, please refer to the datasheets listed below.

DS3231S

AT24C32

Warning

These modules normally come with a 200Ω resistor soldered next to the IN4148 Zener diode, as you can see in the image below.

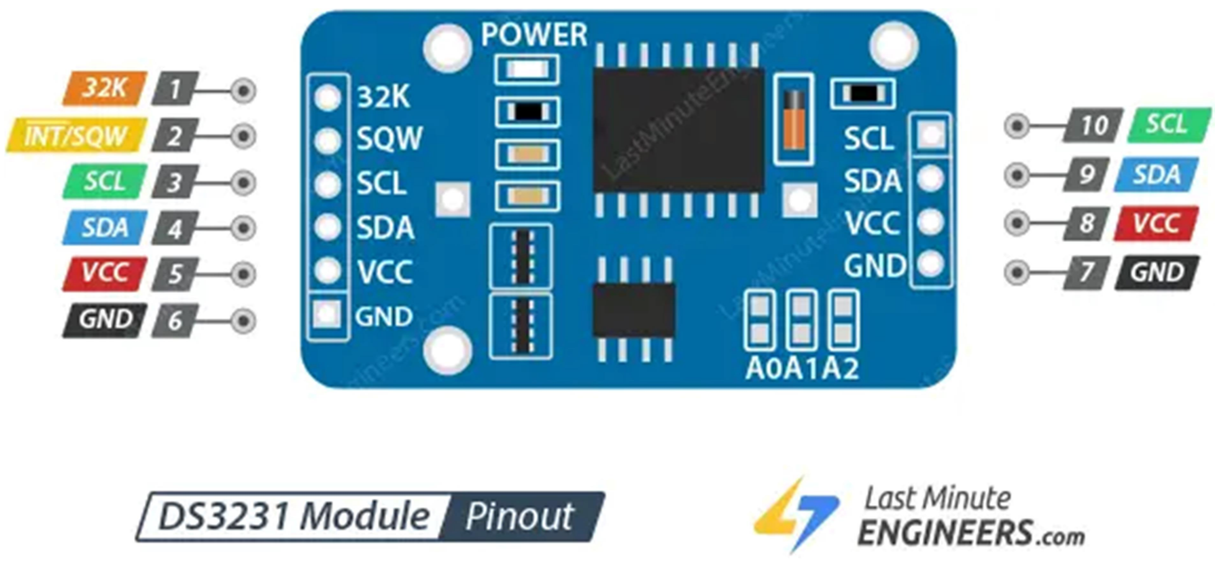


The resistor and diode form a simple charging circuit designed for use with LIR2032 rechargeable batteries.

Be aware that some DS3231 modules come with a non-rechargeable CR2032 battery. If this is the case, you must remove the resistor because attempting to charge a non-rechargeable battery can cause serious damage.

DS3231 RTC Module Pinout

The DS3231 RTC module has 6 pins in total. The pinout is as follows:



32K pin outputs a stable (temperature compensated) and accurate reference clock.

INT/SQW pin provides either an interrupt signal (due to alarm conditions) or a square-wave output at either 1Hz, 4kHz, 8kHz, or 32kHz.

SCL is a serial clock pin for the I2C interface.

SDA is a serial data pin for the I2C interface.

VCC provides power to the module. You can connect it to a 3.3 to 5 volt power supply.

GND is the ground pin.

Wiring a DS3231 RTC module to an Arduino

Let's connect the RTC to the Arduino.

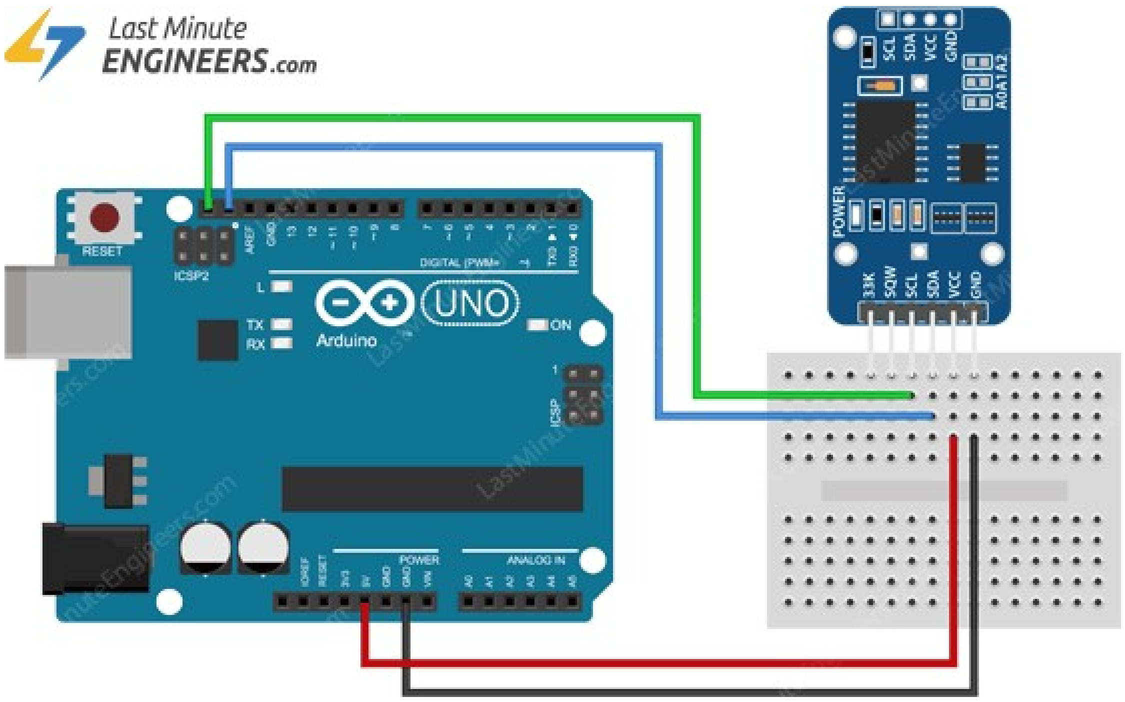
Connections are straightforward. Begin by connecting the VCC pin to the Arduino's 5V output and the GND pin to ground.

Now we are left with the pins that are used for I2C communication. Note that each Arduino board has different I2C pins that must be connected correctly. On Arduino boards with the R3 layout, the SDA (data line) and SCL (clock line) are on the pin headers close to the AREF pin. They are also referred to as A5 (SCL) and A4 (SDA).

Check out the table below for quick reference.

	SCL	SDA
Arduino Uno	A5	A4
Arduino Nano	A5	A4
Arduino Mega	21	20
Leonardo/Micro	3	2

The diagram below shows how to connect everything.

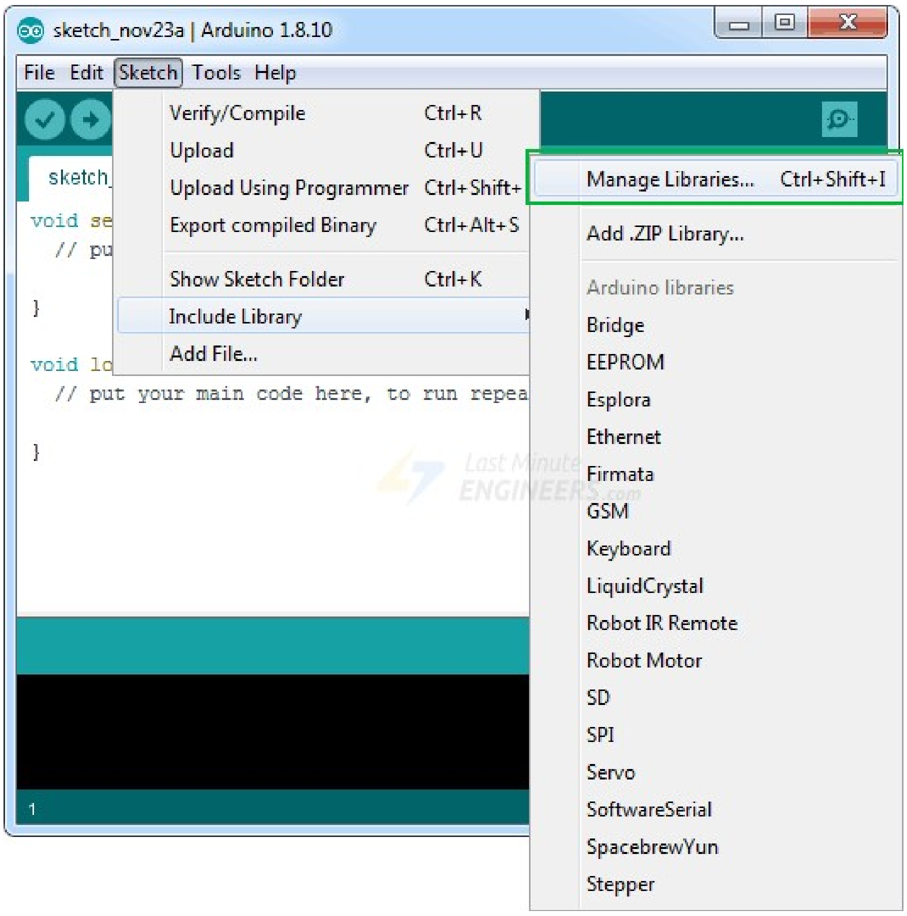


Installing uRTCLib Library

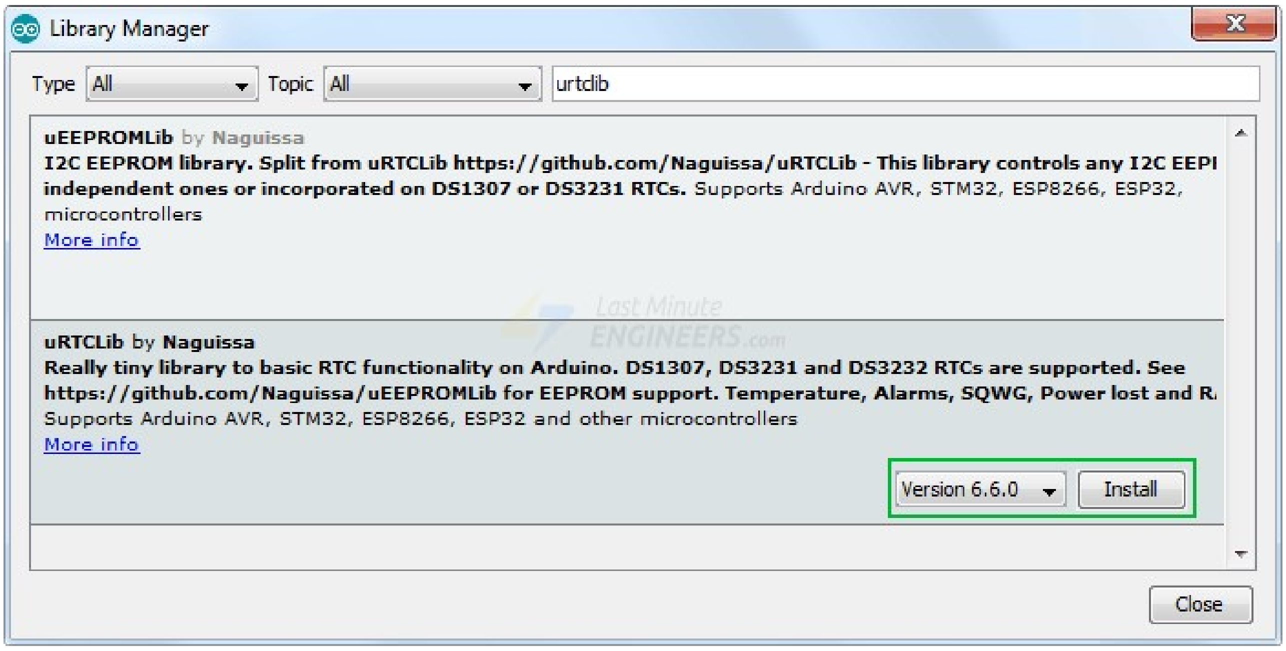
It takes a lot of effort to communicate with an RTC module. Fortunately, the uRTCLib library was created to hide all of the complexities, allowing us to issue simple commands to read the RTC data.

It is a simple yet powerful library that also supports time-of-day alarms and programming the SQW output, which is not supported by many RTC libraries.

To install the library, navigate to Sketch > Include Library > Manage Libraries… Wait for the Library Manager to download the library index and update the list of installed libraries.



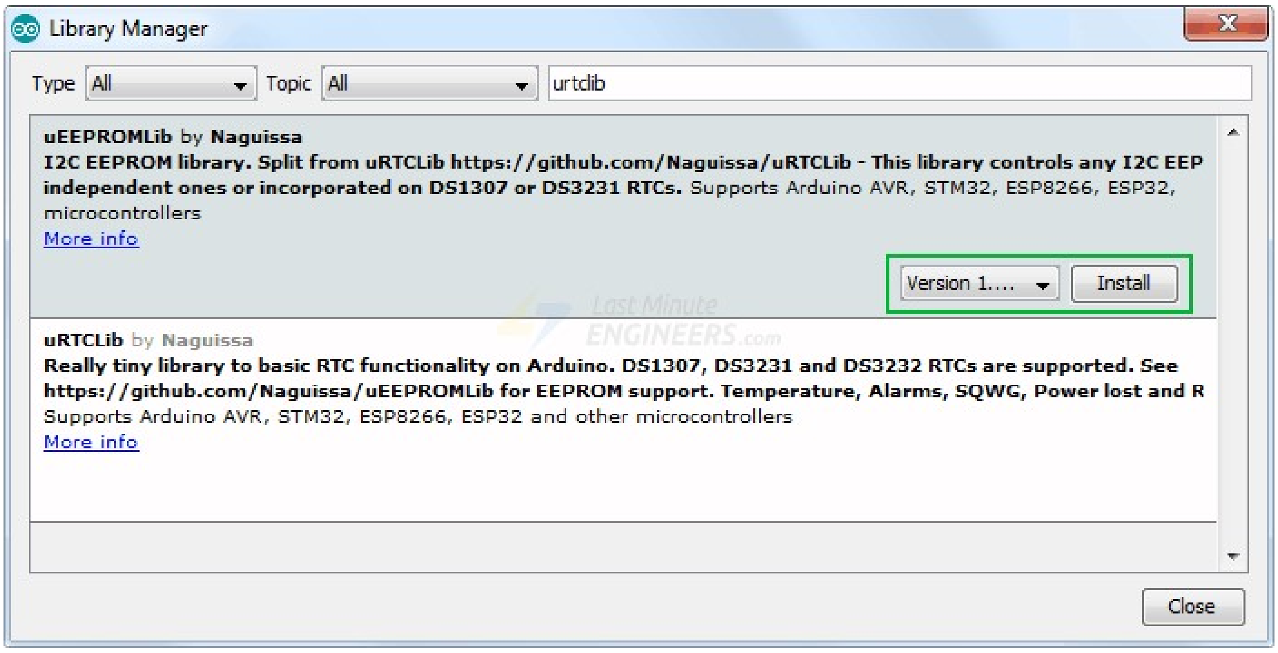
Filter your search by entering ‘urtclib’. Look for uRTCLib by Naguissa. Click on that entry and then choose Install.



uEEPROMLib by Naguissa
I2C EEPROM library. Split from uRTCLib https://github.com/Naguissa/uRTCLib - This library controls any I2C EEP
independent ones or incorporated on DS1307 or DS3231 RTCs. Supports Arduino AVR, STM32, ESP8266, ESP32,
microcontrollers
More info

uRTCLib by Naguissa
Really tiny library to basic RTC functionality on Arduino. DS1307, DS3231 and DS3232 RTCs are supported. See
https://github.com/Naguissa/uEEPROMLib for EEPROM support. Temperature, Alarms, SQWG, Power lost and R
Supports Arduino AVR, STM32, ESP8266, ESP32 and other microcontrollers
More info

At the end of the tutorial, we've also included code for reading and writing the onboard 24C32 EEPROM. If you're interested, you'll need to install the uEEPROMLib library. Look for 'ueepromlib' and install it as well.



uEEPROMLib by Naguissa
I2C EEPROM library. Split from uRTCLib https://github.com/Naguissa/uRTCLib - This library controls any I2C EEP
independent ones or incorporated on DS1307 or DS3231 RTCs. Supports Arduino AVR, STM32, ESP8266, ESP32,
microcontrollers
More info

uRTCLib by Naguissa
Really tiny library to basic RTC functionality on Arduino. DS1307, DS3231 and DS3232 RTCs are supported. See
https://github.com/Naguissa/uEEPROMLib for EEPROM support. Temperature, Alarms, SQWG, Power lost and R
Supports Arduino AVR, STM32, ESP8266, ESP32 and other microcontrollers
More info

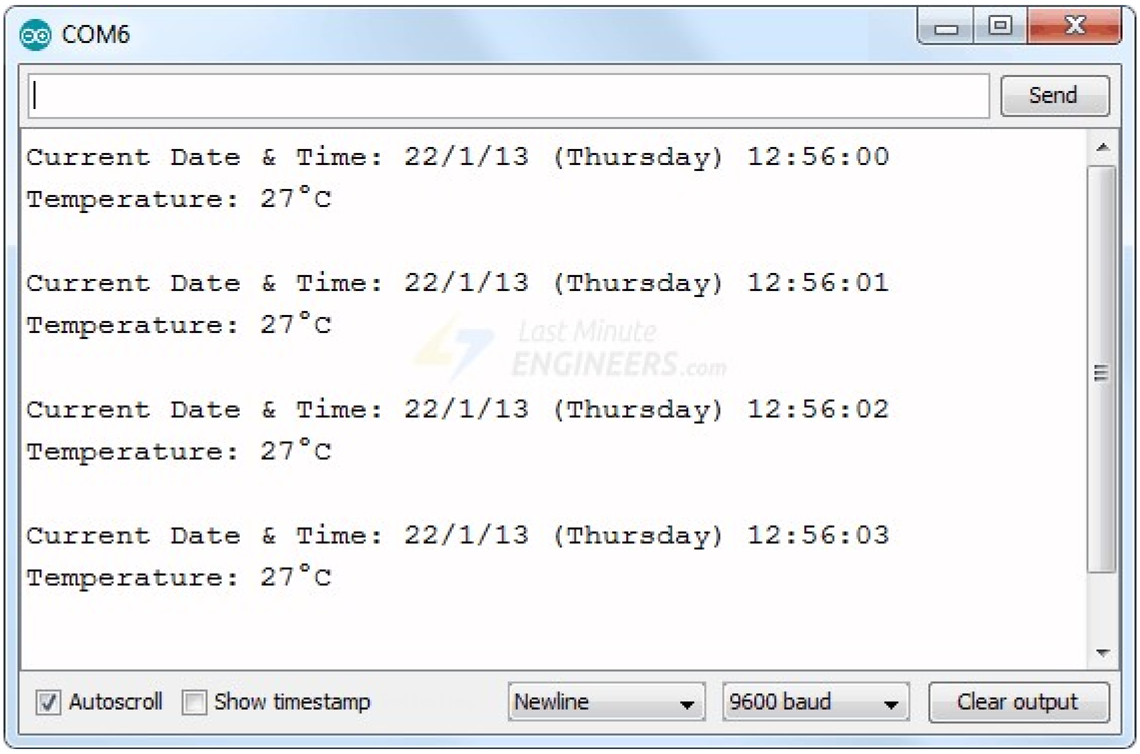
Arduino Code – Reading Date, Time and Temperature

This is a simple sketch for setting/reading the date, time, and temperature from the DS3231 RTC module.

```
#include "Arduino.h"
#include "uRTCLib.h"
```

```
// uRTCLib rtc;
uRTCLib rtc(0x68);

char daysOfTheWeek[7][12] = {"Sunday", "Monday", "Tuesday",
"Wednesday", "Thursday", "Friday", "Saturday"};

void setup() {
  Serial.begin(9600);
  delay(3000); // wait for console opening

  URTCLIB_WIRE.begin();

  // Comment out below line once you set the date & time.
  // Following line sets the RTC with an explicit date & time
  // for example to set January 13 2022 at 12:56 you would call:
   rtc.set(0, 56, 12, 5, 13, 1, 22);
  // rtc.set(second, minute, hour, dayOfWeek, dayOfMonth, month,
year)
  // set day of week (1=Sunday, 7=Saturday)
}

void loop() {
  rtc.refresh();

  Serial.print("Current Date & Time: ");
  Serial.print(rtc.year());
  Serial.print('/');
  Serial.print(rtc.month());
  Serial.print('/');
  Serial.print(rtc.day());

  Serial.print(" (");
  Serial.print(daysOfTheWeek[rtc.dayOfWeek()-1]);
  Serial.print(") ");

  Serial.print(rtc.hour());
  Serial.print(':');
  Serial.print(rtc.minute());
  Serial.print(':');
  Serial.println(rtc.second());

  Serial.print("Temperature: ");
  Serial.print(rtc.temp()  / 100);
  Serial.print("\xC2\xB0");   //shows degrees character
  Serial.println("C");

  Serial.println();
  delay(1000);
}
```

Here's what the output looks like:



```
Current Date & Time: 22/1/13 (Thursday) 12:56:00
Temperature: 27°C

Current Date & Time: 22/1/13 (Thursday) 12:56:01
Temperature: 27°C

Current Date & Time: 22/1/13 (Thursday) 12:56:02
Temperature: 27°C

Current Date & Time: 22/1/13 (Thursday) 12:56:03
Temperature: 27°C
```

Code Explanation:

The sketch begins by including the Arduino.h and uRTCLib.h libraries for communicating with the module. We then create an uRTCLib library object and define the `daysOfTheWeek` 2D character array to store the days information.

To interact with the RTC module, we use the following functions in the setup and loop sections of the code:

begin() function ensures that the RTC module is properly initialized.

set(ss, mm, hh, day, dd, mm, yy) function sets the RTC to an explicit date and time. For example, to set January 13, 2022 at 12:56, you would call: `rtc.set(0, 56, 12, 5, 13, 1, 22);`

refresh() function refreshes data from the HW RTC.

year() function returns the current year.

month() function returns the current month.

day() function returns the current day.

dayOfWeek() function returns the current day of the week (1 to 7). This function is typically used as an index for a 2D character array that stores information about the days.

hour() function returns the current hour.

minute() function returns the current minute.

second() function returns the current seconds.

temp() function returns the current temperature of the 'die'.

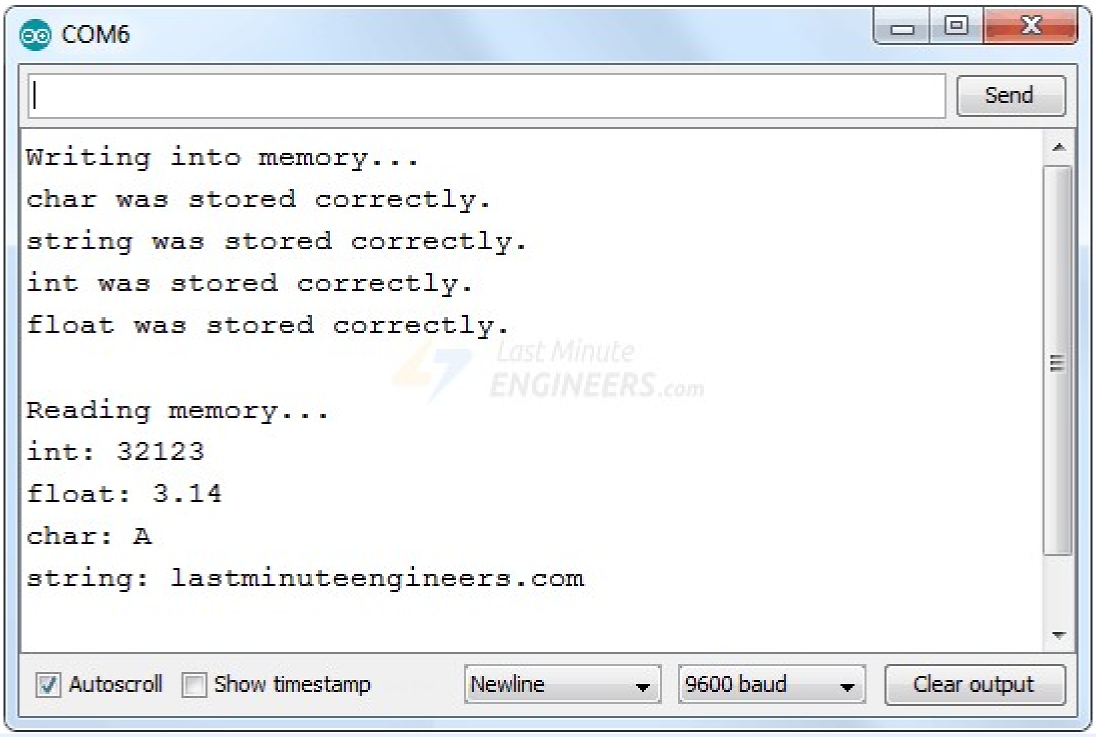
Arduino Code – Read/Write the 24C32 EEPROM

As an added bonus, the DS3231 RTC module includes 32 bytes of Electrically Erasable ROM. Its contents will not be erased even if the device's main power is interrupted.

This is a simple sketch that attempts to write an integer, float, character, and string to the 24C32 EEPROM and then reads them back. This sketch can be extended to save settings, passwords, or just about anything.

```
#include "Arduino.h"
#include "Wire.h"
#include "uEEPROMLib.h"

// uEEPROMLib eeprom;
uEEPROMLib eeprom(0x57);

void setup() {
  Serial.begin(9600);
  delay(2500);

  Wire.begin();

  int inttmp = 32123;
  float floattmp = 3.1416;
  char chartmp = 'A';
  char c_string[] = "lastminuteengineers.com"; //23
  int string_length = strlen(c_string);

  Serial.println("Writing into memory...");

  // Write single char at address
  if (!eeprom.eeprom_write(8, chartmp)) {
  Serial.println("Failed to store char.");
  } else {
  Serial.println("char was stored correctly.");
  }

  // Write a long string of chars FROM position 33 which isn't
aligned to the 32 byte pages of the EEPROM
  if (!eeprom.eeprom_write(33, (byte *) c_string, strlen(c_string)))
{
  Serial.println("Failed to store string.");
  } else {
  Serial.println("string was stored correctly.");
  }

  // Write an int
  if (!eeprom.eeprom_write(0, inttmp)) {
    Serial.println("Failed to store int.");
  } else {
    Serial.println("int was stored correctly.");
  }

  // write a float
  if (!eeprom.eeprom_write(4, floattmp)) {
    Serial.println("Failed to store float.");
  } else {
    Serial.println("float was stored correctly.");
  }

  Serial.println("");
  Serial.println("Reading memory...");

  Serial.print("int: ");
  eeprom.eeprom_read(0, &inttmp);
  Serial.println(inttmp);

  Serial.print("float: ");
  eeprom.eeprom_read(4, &floattmp);
  Serial.println((float) floattmp);

  Serial.print("char: ");
  eeprom.eeprom_read(8, &chartmp);
  Serial.println(chartmp);

  Serial.print("string: ");
  eeprom.eeprom_read(33, (byte *) c_string, string_length);
  Serial.println(c_string);

  Serial.println();
}

void loop() {
}
```

Here's what the output looks like:



```
Writing into memory...
char was stored correctly.
string was stored correctly.
int was stored correctly.
float was stored correctly.

Reading memory...
int: 32123
float: 3.14
char: A
string: lastminuteengineers.com
```

When reading/writing the EEPROM, keep in mind that different data types take up different amounts of memory. For example, a character is one byte, an integer is two bytes, and a float is four bytes.

In other words, if you're storing a character, it can be written to each memory location (0, 1, 2, 3, 4, 5....).

However, if you are storing an integer, you must reserve two memory locations for each value, so you will want to store data in every other memory location, such as (0,2,4,6....).