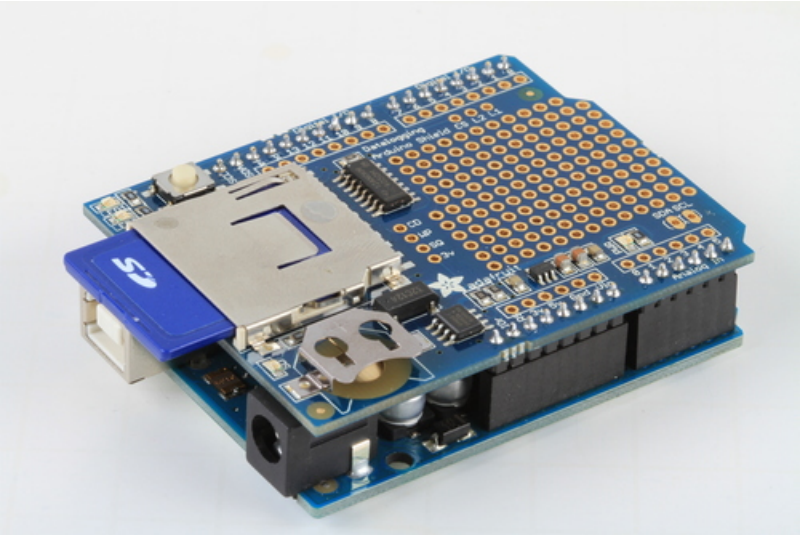
adafruit learning system

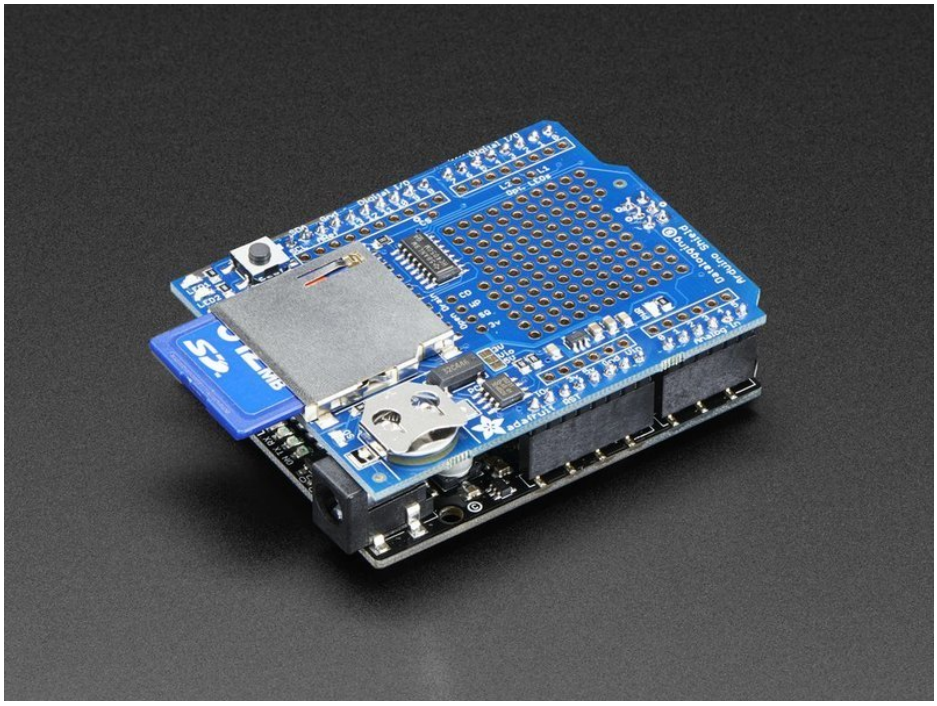
Adafruit Data Logger Shield

Created by Bill Earl



Last updated on 2020-01-20 06:39:09 PM UTC

Overview



Here's a handy Arduino shield: we've had a lot of people looking for a dedicated and well-designed data logging shield. We worked hard to engineer an inexpensive but well-rounded design. This shield makes it easy to add a 'hard disk' with gigabytes of storage to your Arduino!

Our latest version of this popular shield has all the features of the popular original, and is "R3" compatible so you can use it with just about any Arduino or compatible. You can be up and running with it in less than 15 minutes - saving data to files on any FAT16 or FAT32 formatted SD card, to be read by any plotting, spreadsheet or analysis program. This tutorial will also show you how to use two free software programs to plot your data. The included RTC (Real Time Clock) can be used to timestamp all your data with the current time, so that you know precisely what happened when!

The data logger is a reliable, well-rounded and versatile design. It is easily expanded or modified and come well supported with online documentation and libraries

Features:

- SD card interface works with FAT16 or FAT32 formatted cards. Built in 3.3v level shifter circuitry lets you read or write super fast and prevents damage to your SD card
- Real time clock (RTC) keeps the time going even when the Arduino is unplugged. The coin cell battery backup lasts for years
- Included libraries and example code for both SD and RTC mean you can get going quickly
- Prototyping area for soldering connectors, circuitry or sensors.
- Two configurable indicator LEDs
- Onboard 3.3v regulator is both a reliable reference voltage and also reliably runs SD cards that require a lot of power to run
- Uses the "R3 layout" I2C and ICSP/SPI ports so it is compatible with a wide variety of Arduinos and Arduino-compatibles

With this new version you can use it with:

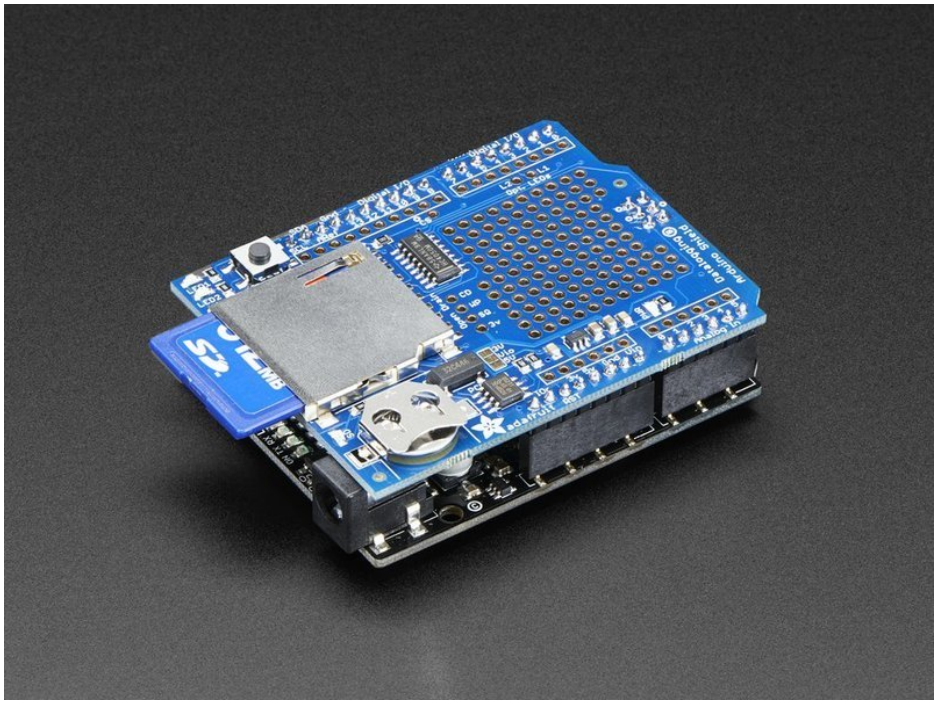
- Arduino UNO or ATmega328 compatible - 4 analog channels at 10 bit resolution, 6 if RTC is not used
- Arduino Leonardo or ATmega32u4 compatible - 12 analog channels at 10 bit resolution
- Arduino Mega or ATmega2560 compatible - 16 analog inputs (10-bit)
- Arduino Zero or ATSAMD21 compatible - 6 analog inputs (12-bit)
- Arduino Due compatible - 12 analog inputs (12-bit)

Of course you can log anything you like, including digital sensors that have Arduino libraries, serial data, bit timings, and more!

Installing the Headers

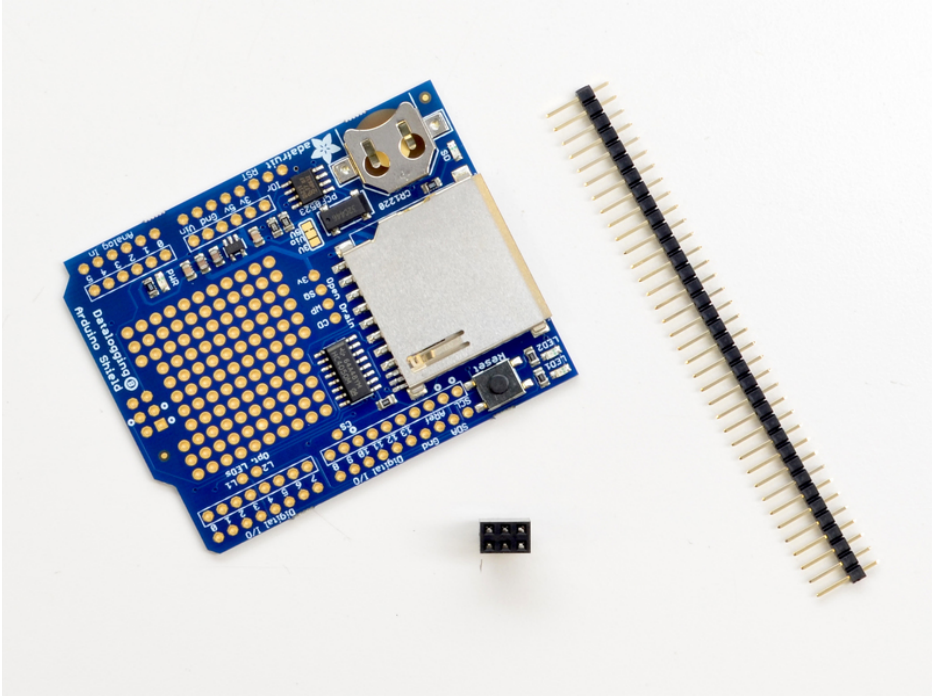
The Adafruit Data Logger shield comes tested assembled with all components and SD socket already on it, but you'll still need need to put headers on so you can plug it into an Arduino

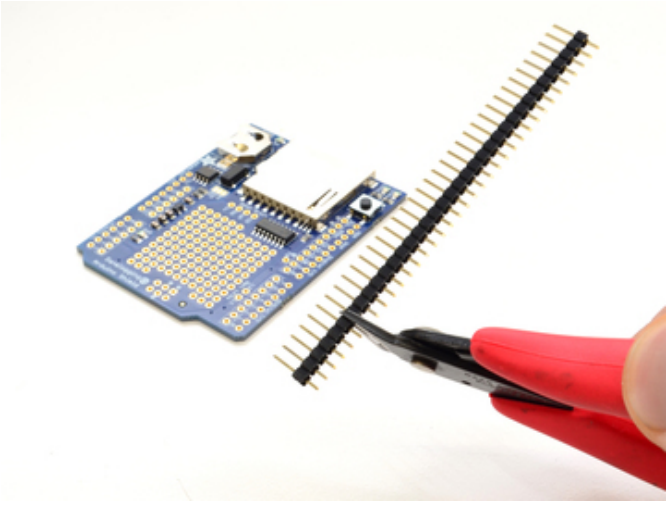
We don't pre-assemble the headers on because there's **two** options! You can either use plain 0.1" male headers (included with the shield) or Arduino Shield Stacking headers (http://adafru.it/85). Both options additionally **require** a 2x3 female header soldered on.



Assembly with male headers

Most people will be happy with assembling he shield with male headers. The nice thing about using these is they don't add anything to the height of the project, and they make a nice solid connection. However, you won't be able to stack another shield on top. Trade offs!





Cut the headers to length:

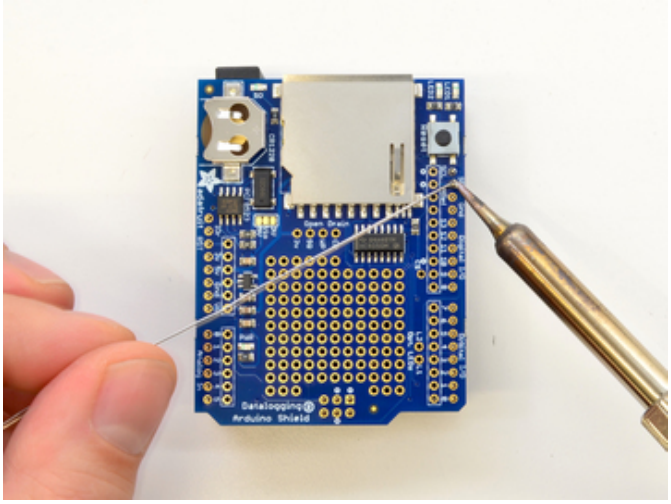
Line the header strip up with the holes on the edge of the shield and cut 4 sections of header strip to fit.

Position the headers:

Insert the header sections - long pins down - into the female headers on your Arduino/Metro. Additionally insert the 2x3 female header into the corresponding pins on the opposite side as the USB.

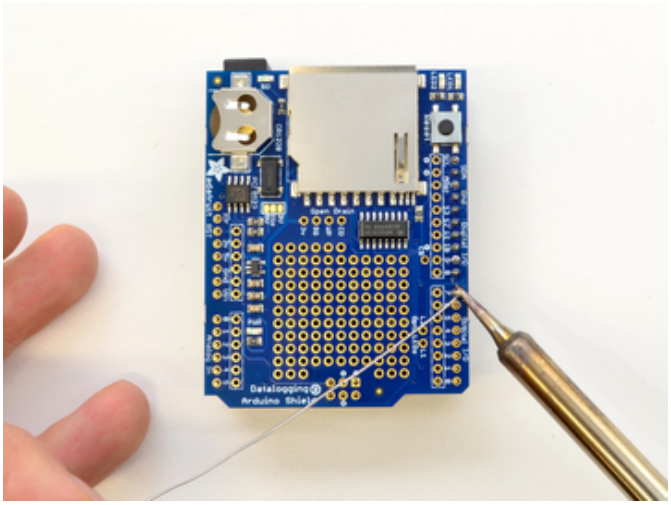
Position the shield:

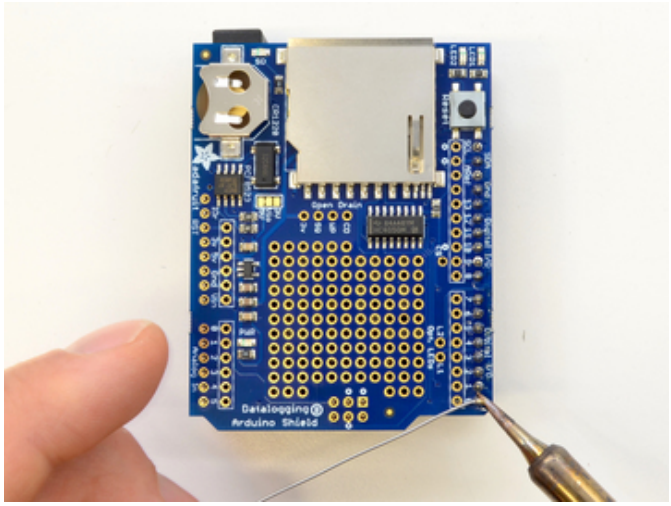
Align the shield with the header pins and press down.

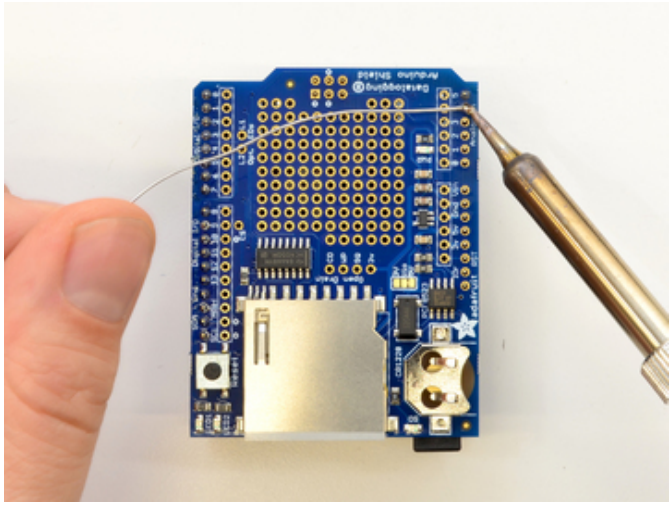


And solder!

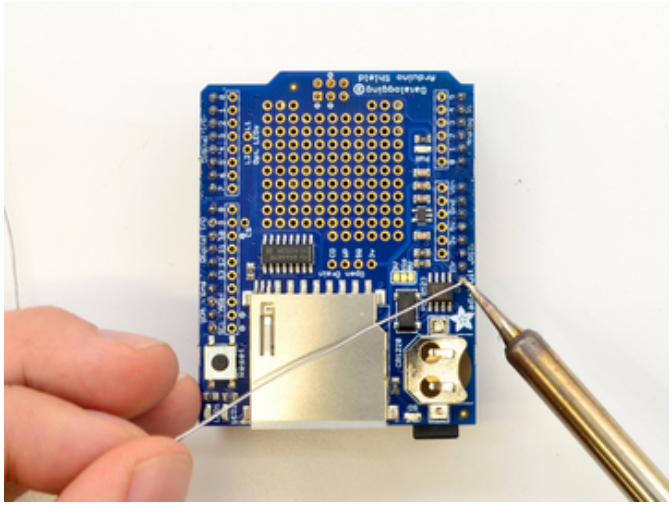
Solder each pin to assure good electrical contact. For tips on soldering, refer to the Adafruit Guide to Excellent Soldering (https://adafru.it/c6b).

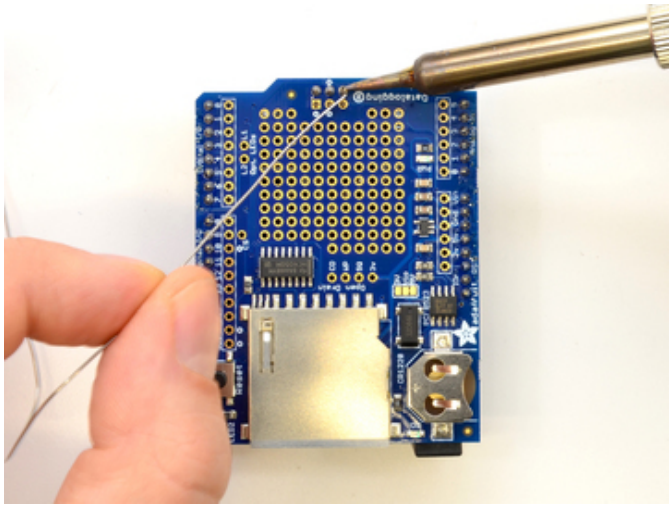


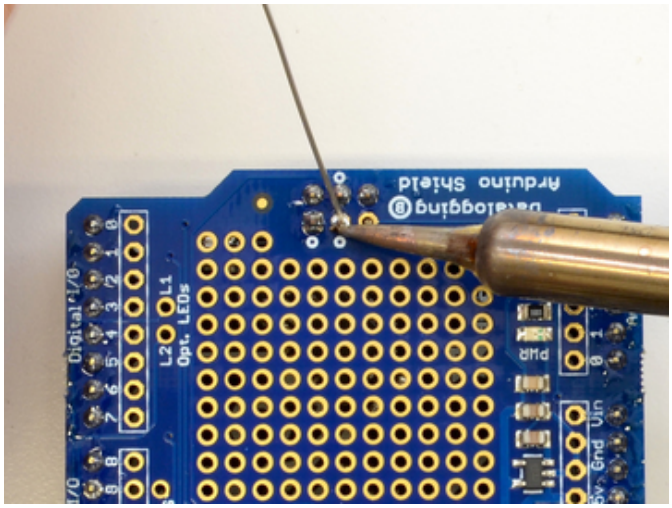


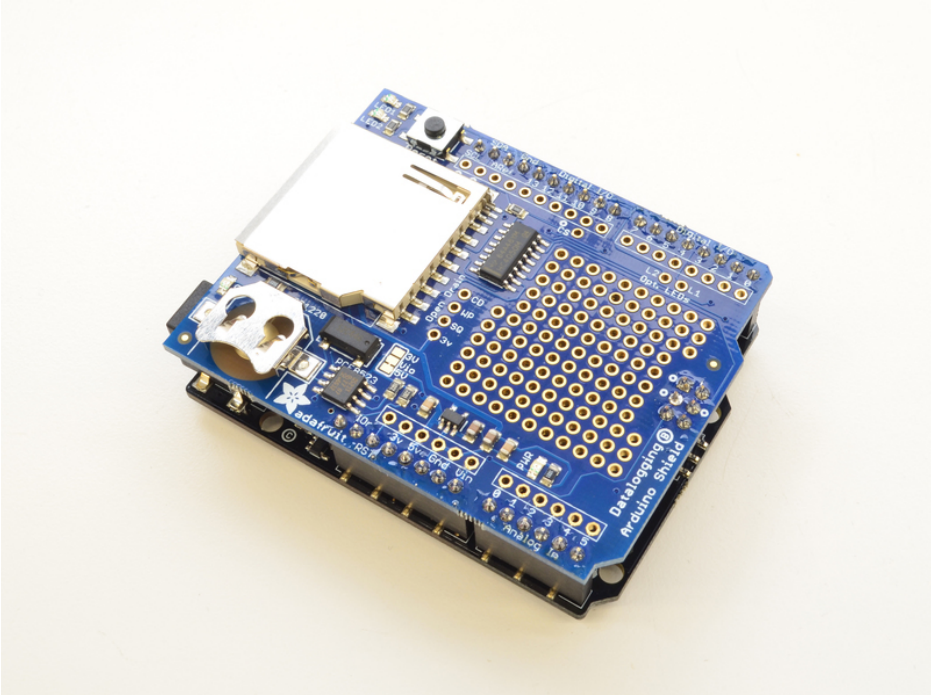


Flip around and solder the other side as well as the 2x3 header






Datalogging
Arduino Shield
Digital I/O
L2
L1
Opt. LEDs
PWR
Gnd
Vin



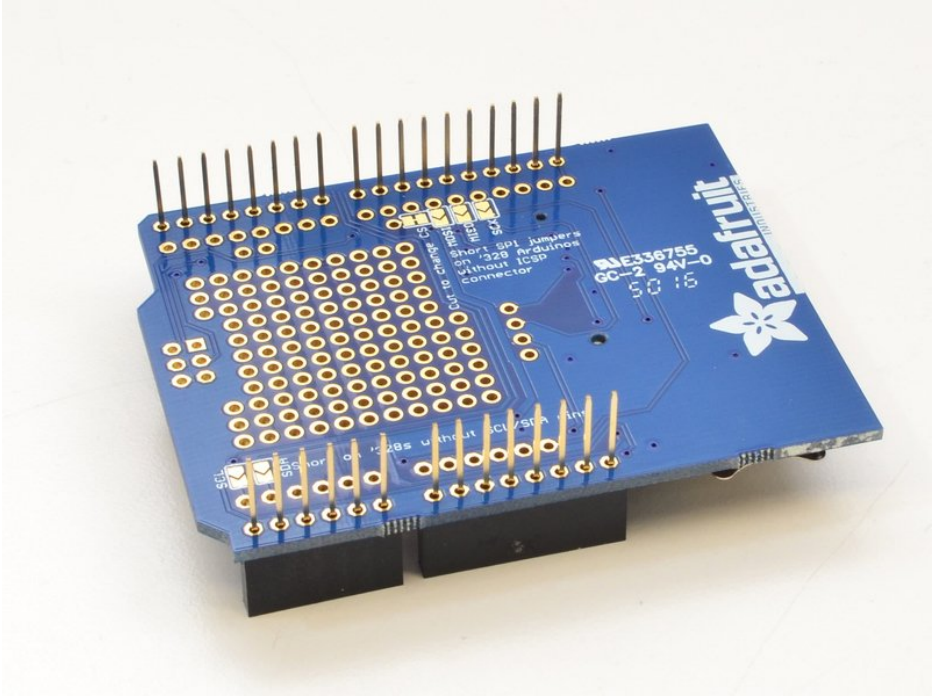
Assembly with Stacking Headers:

Stacking headers give your data logger shield extra flexibility. You can combine it with other shields such as the RGB/LCD Display shield (http://adafru.it/714) to make a compact logging instrument complete with a user interface. You can also stack it with one or more Proto-Shields (http://adafru.it/51) to add even more prototyping space for interfacing to sensors.

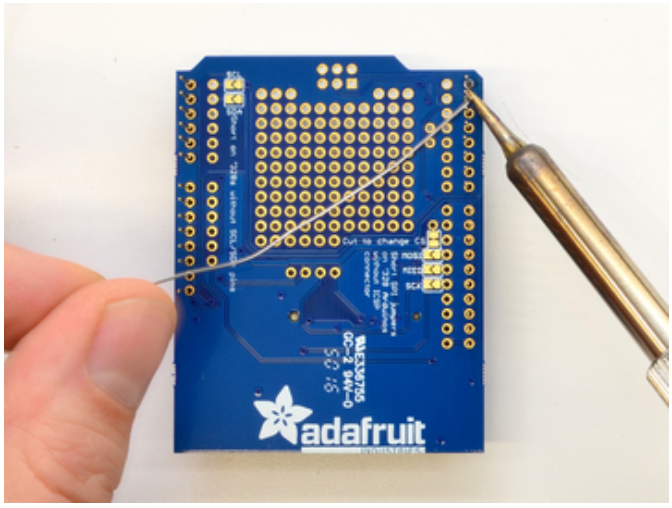
Stacking headers are installed from the top of the board instead of the bottom, so the procedure is a little different than for installing simple male headers.

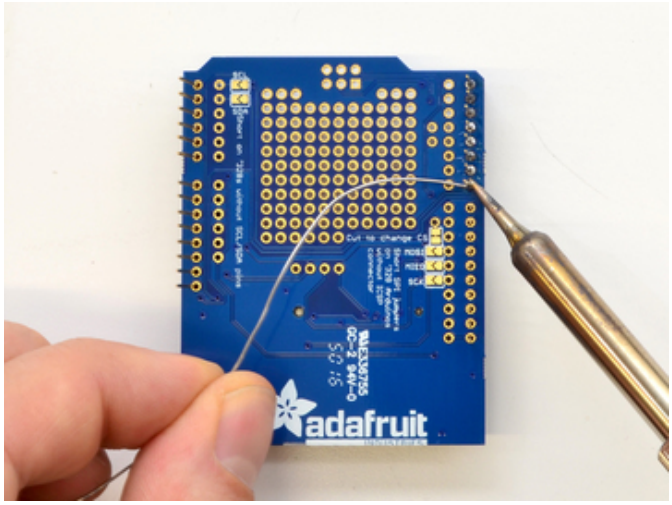
Position the headers:

Insert the headers **from the top** of the shield, then **flip the shield over** and place it on a flat surface. Straighten the headers so that they are vertical.



Be sure to insert the headers from the TOP of the shield so that they can be soldered from the BOTTOM.



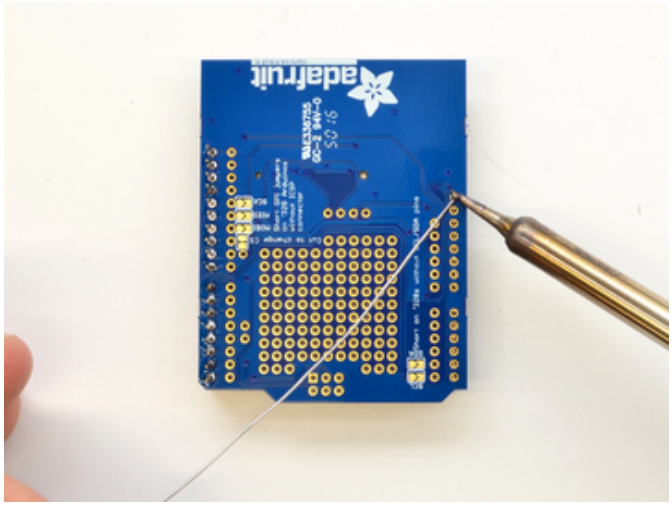


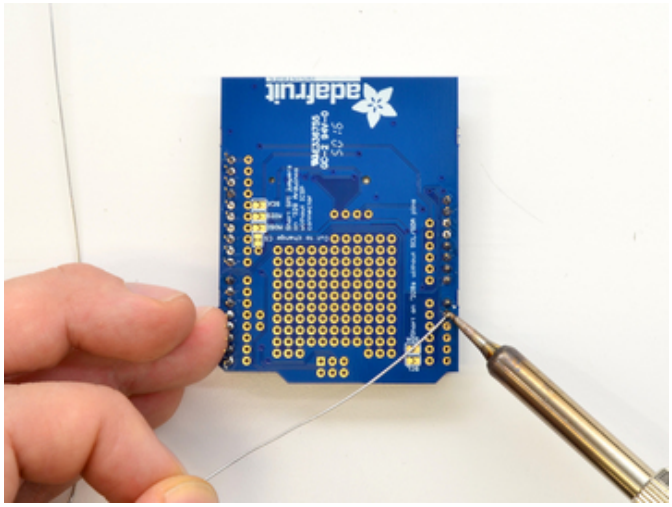
And solder!

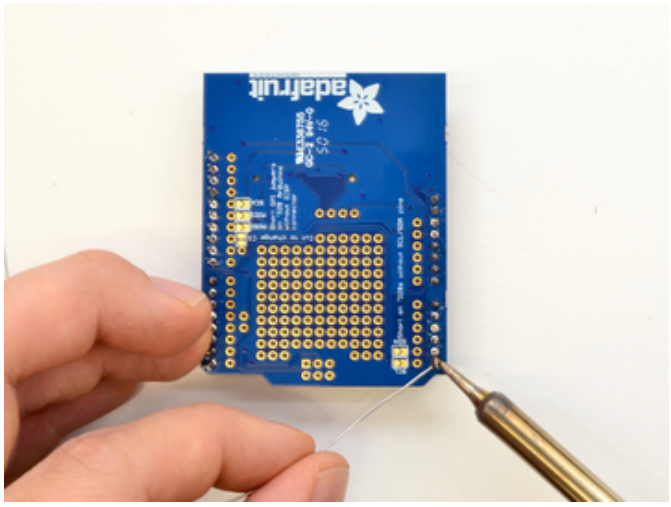
Solder each pin for a solid electrical connection.

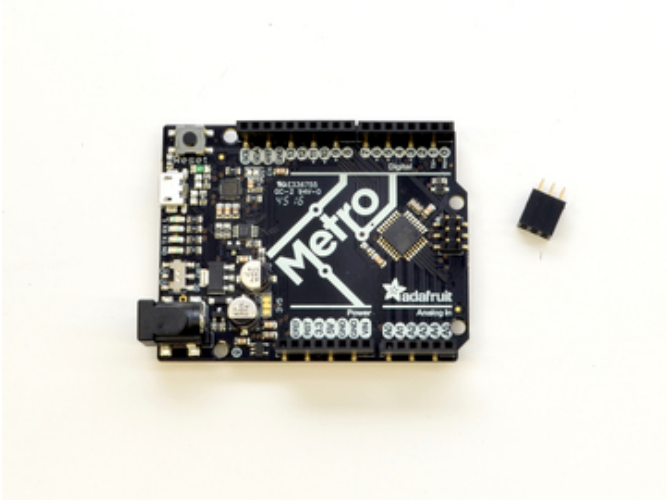
Tip: Solder one pin from each header section. If any of them are crooked, simply re-heat the one solder joint and straighten it by hand. Once all headers are straight, continue soldering the rest of the pins.

Flip and solder the other side



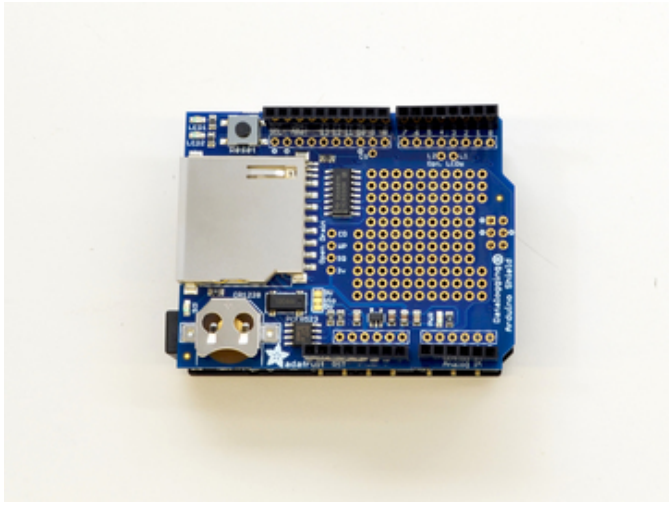


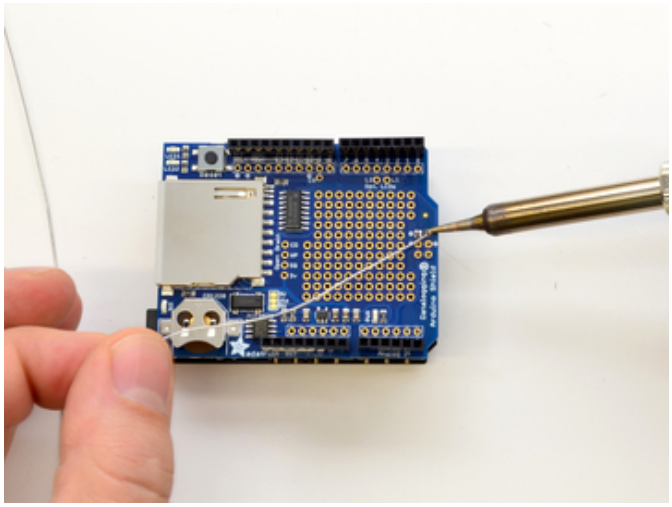


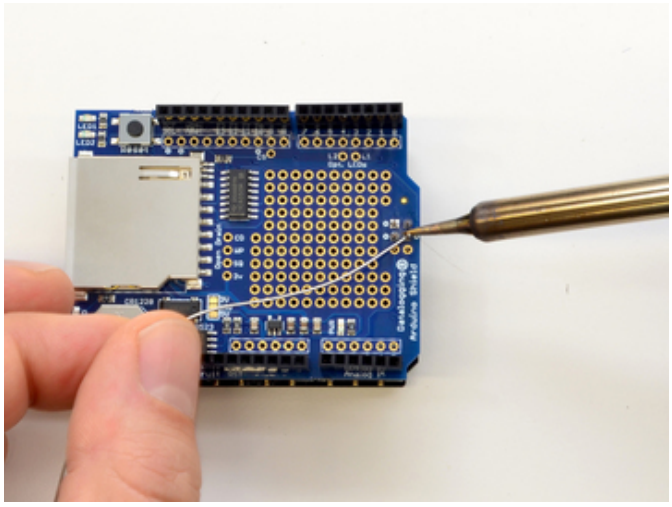


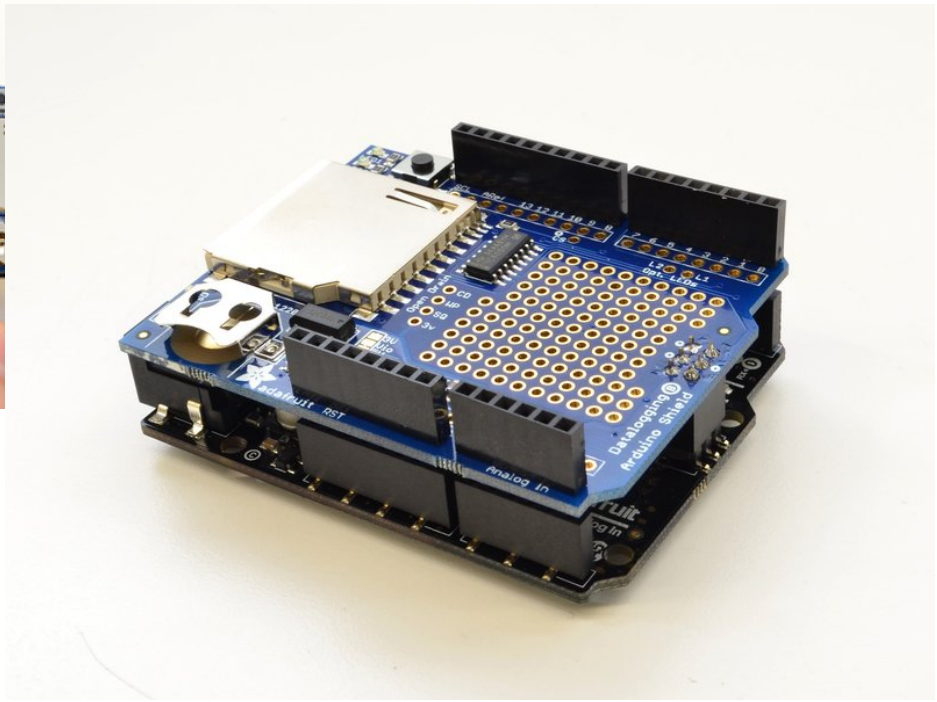
Place the 2x3 female header on to the Arduino/Metro

Place the board on the Metro and solder the 2x3 header



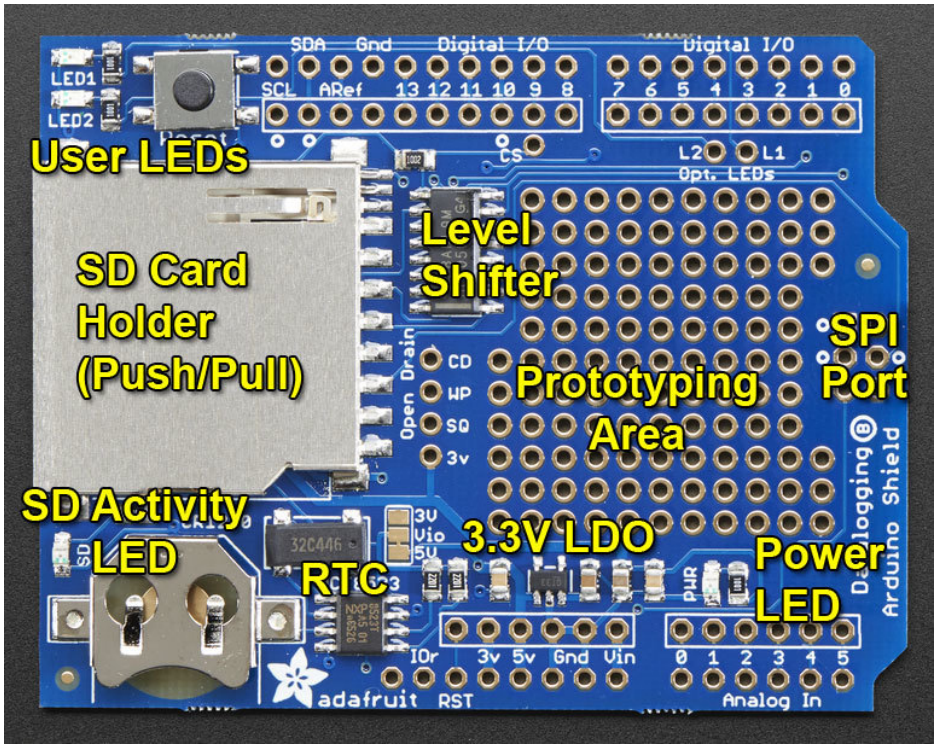




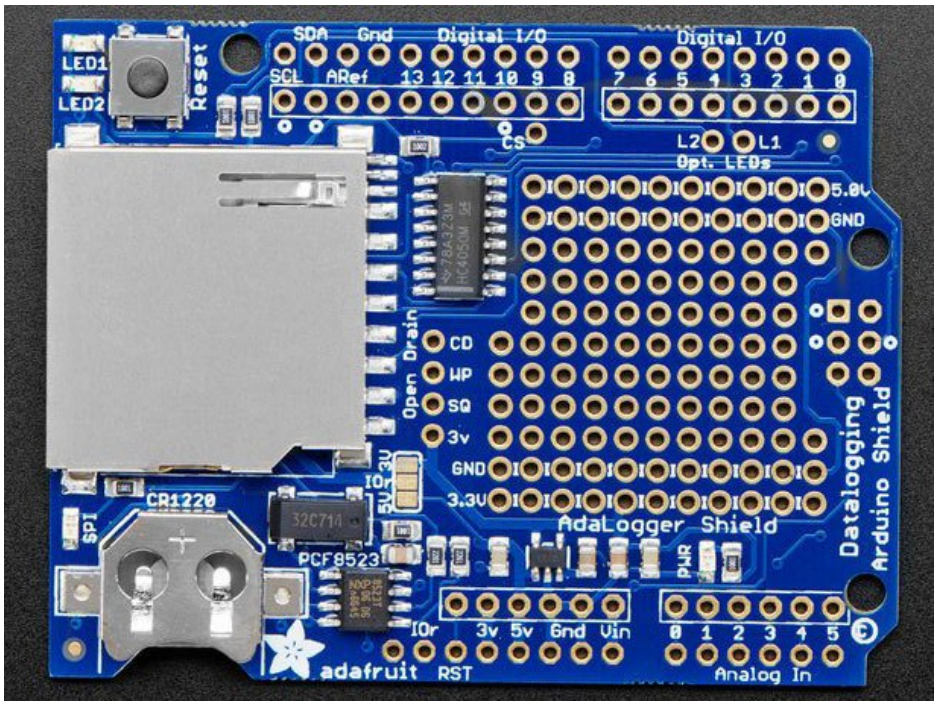

Analog In
Datalogging
Arduino Shield

Shield Overview

The datalogger shield has a few things to make it an excellent way to track data. Here's a rough map of th shield:



Our latest version adds power rails for 5V, 3.3V and Ground:



SD Card

The big SD card holder can fit any SD/MMC storage up to 32G and and small as 32MB (Anything formatted FAT16 or FAT32) If you have a MicroSD card, there are low cost adapters which will let you fit these in. SD cards are tougher to lose than MicroSD, and there's plenty of space for a full size holder.

Simply Push to insert, or Pull to remove the card from this slot

The **SD Activity LED** is connected to the clock pin, it will blink when data goes over SPI, which can help you detect when its ok to remove or insert the SD card or power down the Arduino.

The **Level Shifter** moves all signals from 3.3 or 5V down to 3.3V so you can use this shield with *any* Arduino safely and not damage cards. Cheaper shields use resistors to level shift, but this doesn't work well at high speed or at all voltage levels!

Real Time Clock

This is the time-keeping device. It includes the 8-pin chip, the rectangular 32KHz crystal and a battery holder

The battery holder must contain a battery in order for the RTC to keep track of time when power is removed from the Arduino! Use any CR1220 compatible coin cell

CR1220 12mm Diameter - 3V Lithium Coin Cell Battery

$0.95
IN STOCK

Add To Cart

3.3V Power Supply

An on-board 3.3V LDO (low drop-out type) regulator keeps the shield's 3V parts running smoothly. Some old Arduinos did not have a full 3.3V regulator and writing to an SD card could cause the Arduino to reboot. To maintain compatibility we just keep it there. There's also a green PWR (Power) good LED to the right

User LEDs

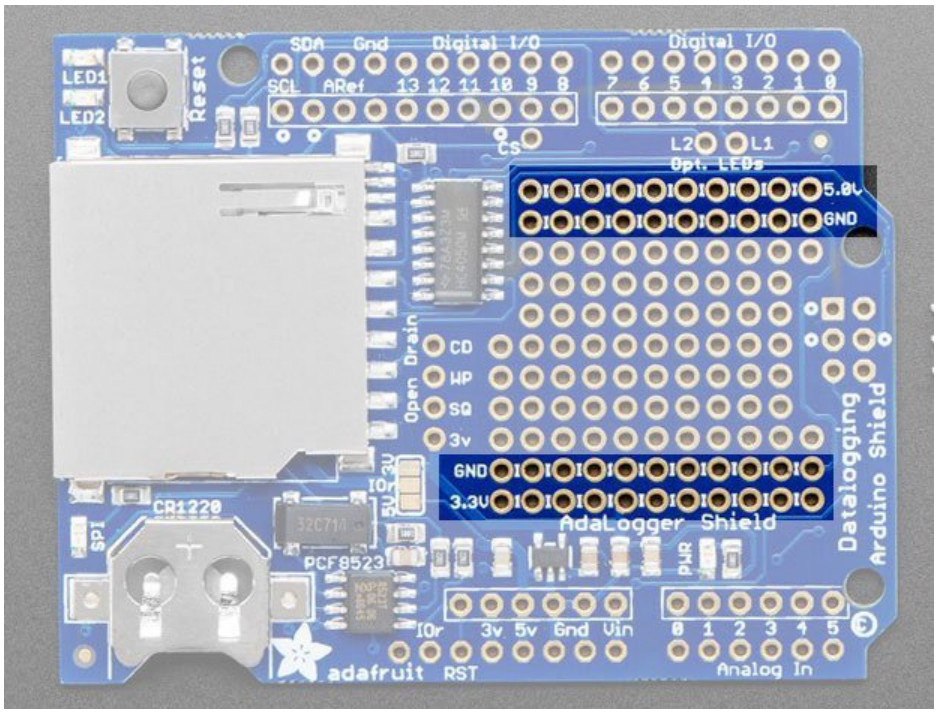
We have two user-configuratble LEDs. Connect a wire from any Arduino pin to **L1** or **L2** marked pads and pull high to turn on **LED1** or **LED2**

The reset button to the right of the LEDs, will reset the entire Arduino, handy for when you want to restart the board

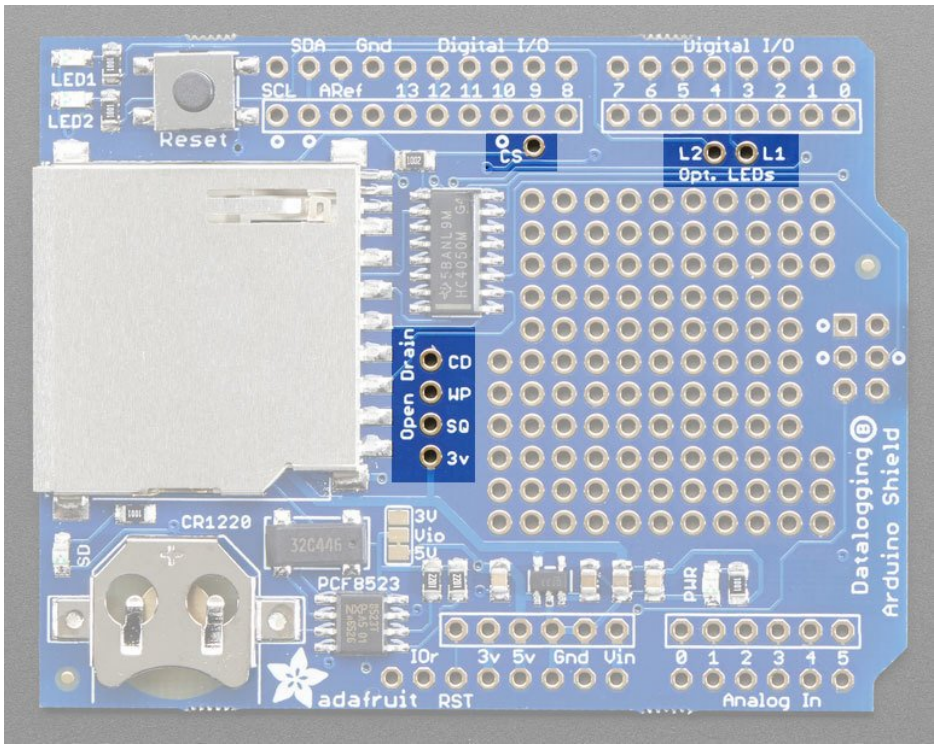
Prototyping Area

The big middle section is filled with 0.1" grid prototyping holes so you can customize your shield with sensors or other circuitry.

The top two and bottom two rows of proto holes are power rails.



Breakout Pads



We also have some extra breakouts shown above, around the breakout board area.

To the right of the SD card holder:

- **CD** - this is the card detect pad on the SD card. When this is connected to ground, an SD card is inserted. It is open-drain, use a pullup (either physical resistor or enabled in software)
- **WP -** this is the Write Protect pad on the SD card, you can use this to detect if the write-protect tab is on the card by checking this pin. It is open-drain, use a pullup (either physical resistor or enabled in software)
- **SQ** - this is the optional Squarewave output from the RTC. You have to send the command to turn this on but its a way of optionally getting a precision squarewave. We use it primarily for testing. The output is open drain so a pullup (either physical resistor or enabled in software)
- **3V** - this is the 3V out of the regulator. Its a good quality 3.3V reference which you may want to power sensors. Up to 50mA is available

Near Digital #10

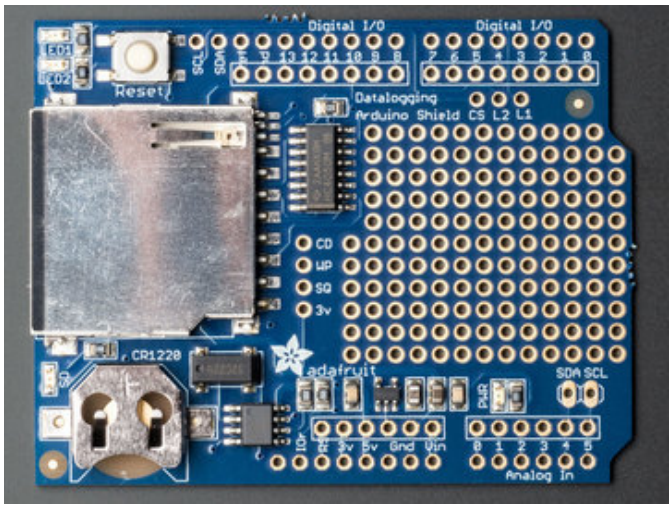
- **CS** - this is the **Chip Select** pin for the SD card. If you need to cut the trace to pin 10 because it is conflicting, this pad can be soldered to any digital pin and the software re-uploaded

Near Digital #3 and #4

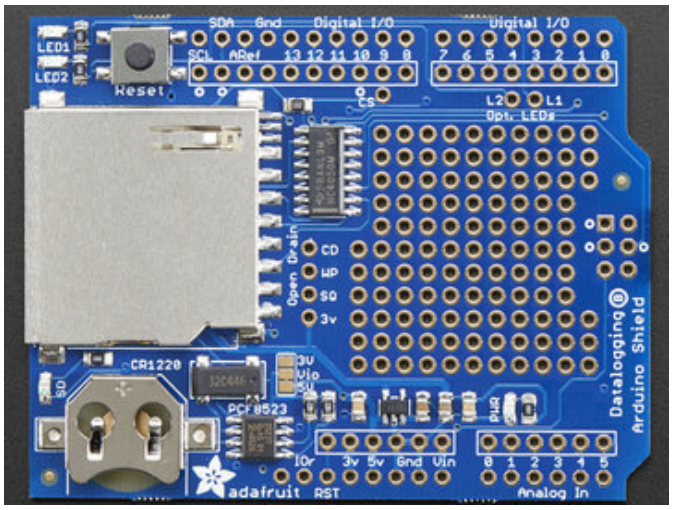
- **L2** and **L1** - these are optional user-LEDs. Connect to any digital pin, pull high to turn on the corresponding LED. The LEDs already have 470 ohm resistors in series.

As of revision B of the Datalogger shield, we've moved away from using digital pins 10, 11, 12, 13 for SPI and A4, A5 for I2C. We now use the 2x3 ICSP header, which means that you don't need special customized I2C or SPI libraries to use with Mega or Leonardo or Zero (or any other future type) of Arduino!

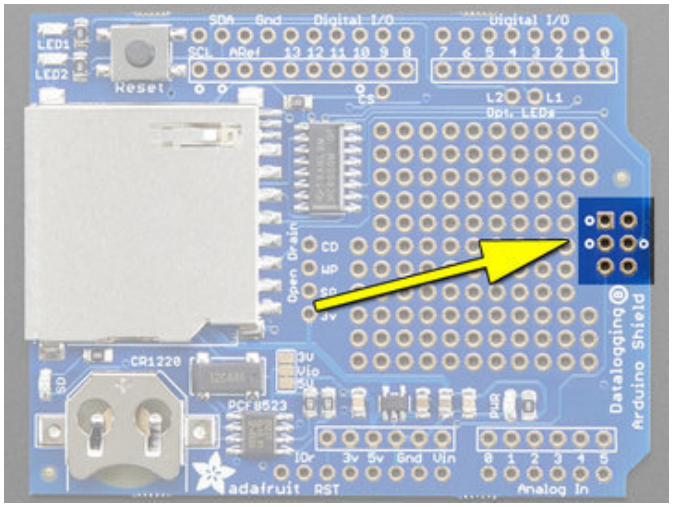
Which version do I have?



This is the older Datalogger shield. In particular, note that the prototyping area is completely full of 0.1" spaced holes



This is the "R3 compatible" Datalogger. Note that it has a smaller prototyping area and that there is a 2x3 SPI header spot on the right



Older Shield Pinouts

On the older shields, the pinout was *fixed* to be:

- **Digital #13** - SPI clock
- **Digital #12** - SPI MISO
- **Digital #11** - SPI MOSI
- **Digital #10** - SD Card chip select (can cut a trace to re-assign)
- **SDA** connected to **A4**
- **SCL** connected to **A5**

The RTC (DS1307) I2C logic level was fixed to 5V

Rev B Shield Pinouts

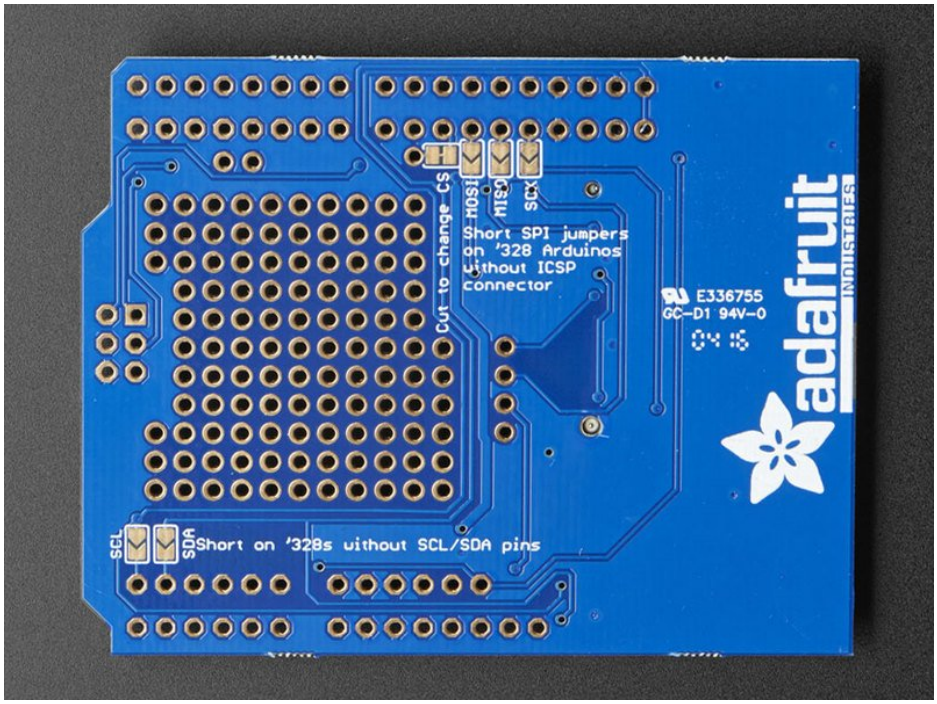
- **ICSP SCK** - SPI clock
- **ICSP MISO** - SPI MISO
- **ICSP MOSI** - SPI MOSI
- **Digital #10** - SD Card chip select (can cut a trace to re-assign)
- **SDA** *not* connected to **A4**

- **SCL** *not* connected to **A5**

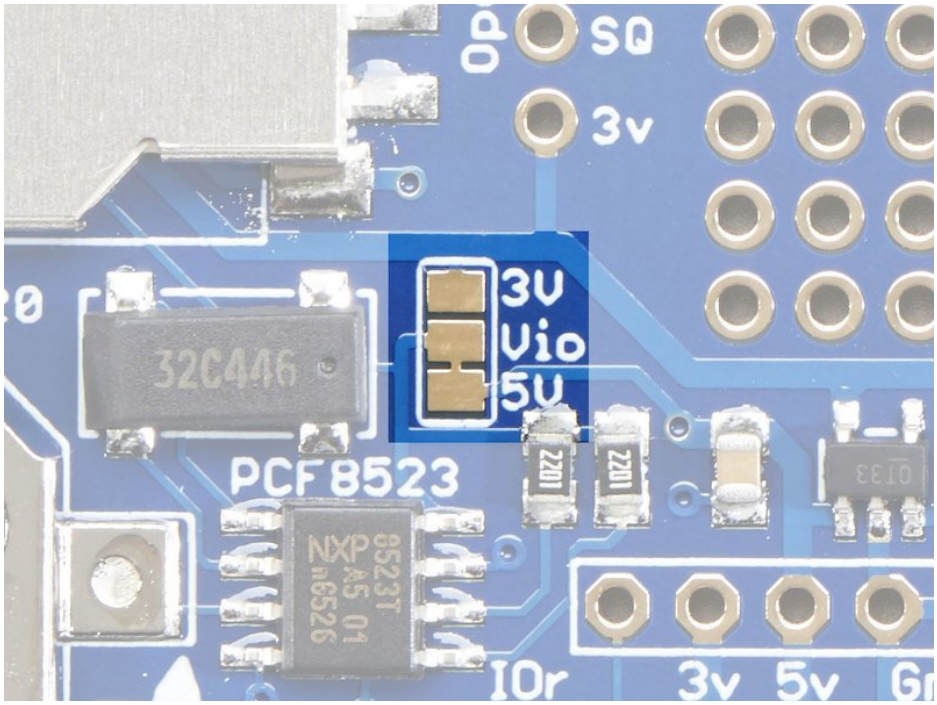
The RTC (PCF8523) logic level can be 3V or 5V

On an UNO, note that Digital #13 is the same as ICSP SCK, #12 is ICSP MISO, #11 is ICSP MOSI, SDA is tied to A4 and SCL is A5. However, that is only true on the UNO! Other Arduino's have different connections. Since the shield no longer makes the assumption it's on an UNO, it is the most cross-compatible shield.

On the bottom of the Rev B shield, you can see that if you have an older Arduino where there is no ICSP 2x3 header, and no SDA/SCL pins, you can short the solder jumpers closed.

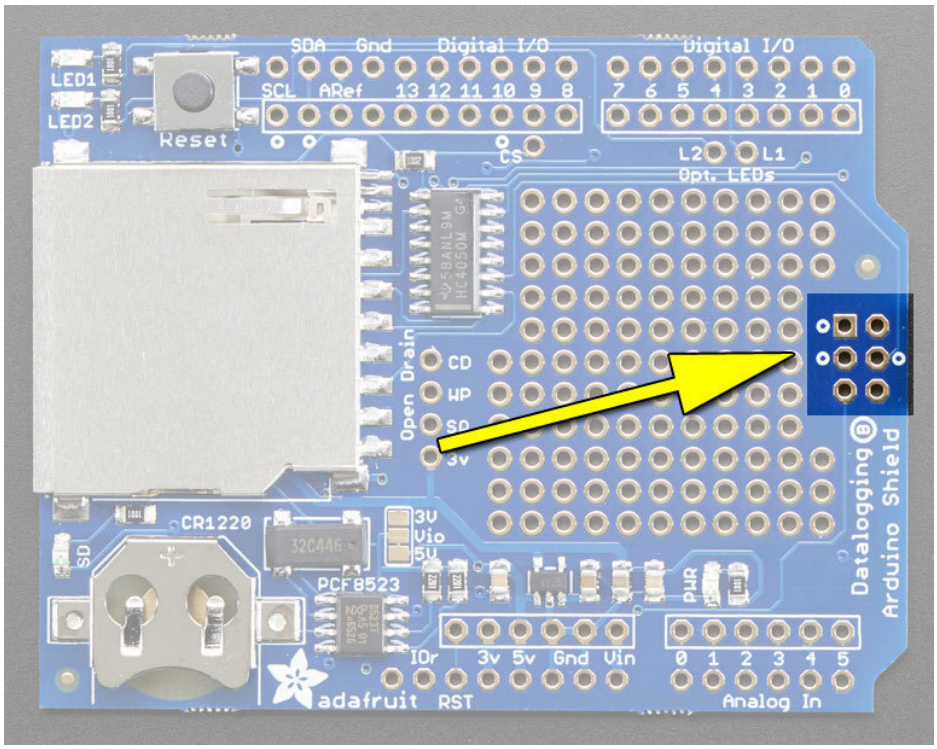


If you are using the shield with a 3.3V logic Arduino, you may want to change the **Vio** jumper. This is what the 10K pullups for I2C are pulled up to. Honestly, the pullups are very weak so if you forget, it's not a big deal. But if you can, cut the small trace between the center pad and 5V and solder the other side so that Vio is connected to 3V


SQ
3v
3V
Vio
5V
32C446
PCF8523
NXP
8523T
A5 01
6526
220
220
IOr
3v
5v

This is ONLY required if you have the older Datalogger shield which does not have the SPI port connection.

This is ONLY required if you are using a Leonardo or Mega with the older Datalogger shield!

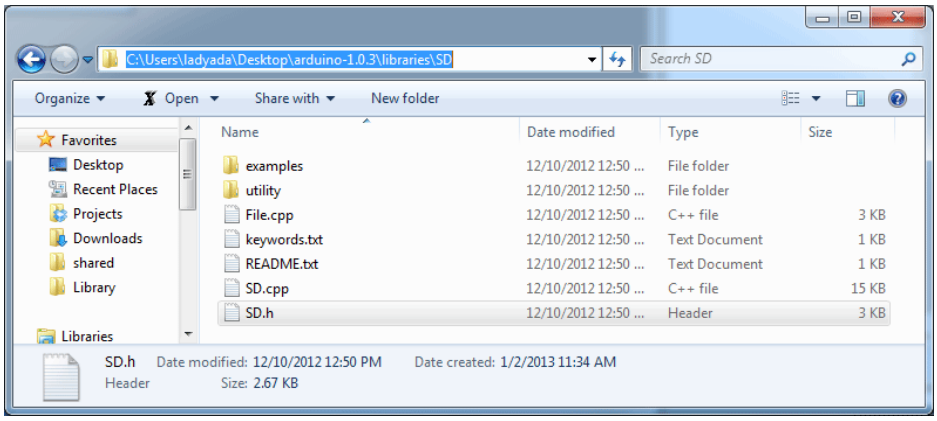


If your shield looks like the above, and has the 2x3 pin header on the right, *skip this page!*

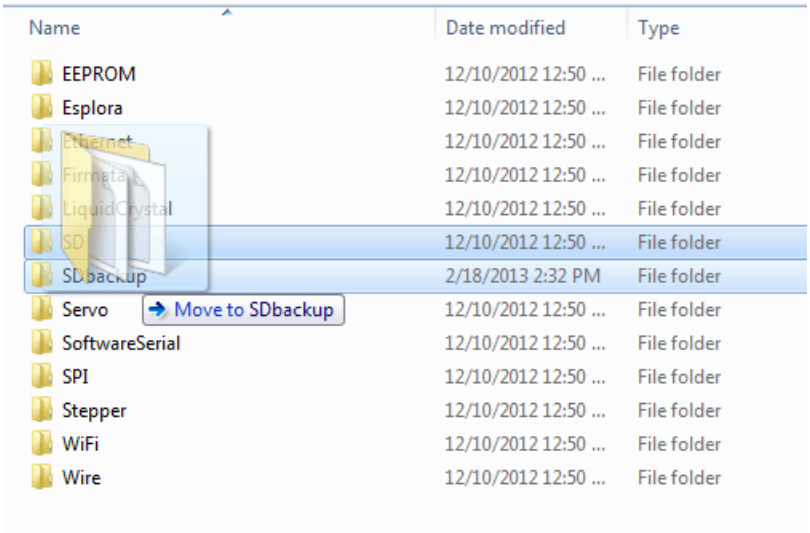
If your shield does not have the 2x3 pin header section **and** you are using a Mega or Leonardo (e.g. not UNO-compatible) then you can keep reading!

If you are using an Leonardo or Mega with the older datalogging shield, you will have to replace the existing SD card library to add 'SD card on any pin' support. **If you have an Uno/Duemilanove/Diecimila, this is not required. If you have a rev B shield, this is also not required!**

First, find the "core libraries" folder - if you are using Windows or Linux, it will be in the folder that contains the **Arduino** executable, look for a **libraries** folder. Inside you will see an **SD** folder (inside that will be **SD.cpp SD.h** etc)



Ourside the **libraries** folder, make a new folder called **SDbackup**. Then drag the **SD**folder into **SDbackup**, this will 'hide' the old **SD** library without deleting it. *Note that SDBackup must be **outside** of the libraries folder in order to effectively 'hide' the SD library.*



Now we'll grab the new SD library, visit https://github.com/adafruit/SD (https://adafru.it/aP6) and click the**ZIP** download button, or click the button below

https://adafru.it/cxl

https://adafru.it/cxl

Uncompress and rename the uncompressed folder **SD**. Check that the **SD** folder contains **SD.cpp** and **SD.h**

Place the **SD** library folder your sketchbook libraries folder. You may need to create the libraries subfolder if its your first library. For more details on how to install libraries, check out our ultra-detailed tutorial at (https://adafru.it/aYM)http://learn.adafruit.com/adafruit-all-about-arduino-libraries-install-use (https://adafru.it/aYM)

Using the SD Library with the Mega and Leonardo

Because the Mega and Leonardo do not have the same hardware SPI pinout, you need to specify which pins you will be using for SPI communication with the card. For the data logger shield, these will be pins 10, 11, 12 and 13. Find the location in your sketch where SD.begin() is called (like this):

```
// see if the card is present and can be initialized:
if (!SD.begin(chipSelect)) {
```

and change it to add these pin numbers as follows:

```
// see if the card is present and can be initialized:
if (!SD.begin(10, 11, 12, 13)) {
```

cardinfo

The cardinfo sketch uses a lower level library to talk directly to the card, so it calls card.init() instead of SD.begin().

```
// we'll use the initialization code from the utility libraries
// since we're just testing if the card is working!
while (!card.init(SPI_HALF_SPEED, chipSelect)) {
```

When calling card.init(), you must change the call to specify the SPI pins, as follows:

```
// we'll use the initialization code from the utility libraries
// since we're just testing if the card is working!
while (!card.init(SPI_HALF_SPEED, 10, 11, 12, 13)) {
```

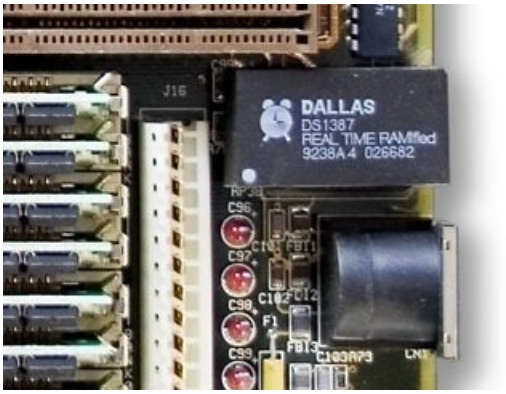
What is a Real Time Clock?

When logging data, it's often really really useful to have timestamps! That way you can take data one minute apart (by checking the clock) or noting at what time of day the data was logged.

The Arduino does have a built-in timekeeper called **millis()** and theres also timers built into the chip that can keep track of longer time periods like minutes or days. So why would you want to have a separate RTC chip? Well, the biggest reason is that **millis()** only keeps track of time *since the Arduino was last powered* - that means that when the power is turned on, the millisecond timer is set back to 0. The Arduino doesnt know its 'Tuesday' or 'March 8th' all it can tell is 'Its been 14,000 milliseconds since I was last turned on'.

OK so what if you wanted to set the time on the Arduino? You'd have to program in the date and time and you could have it count from that point on. But if it lost power, you'd have to reset the time. Much like very cheap alarm clocks: every time they lose power they blink **12:00**

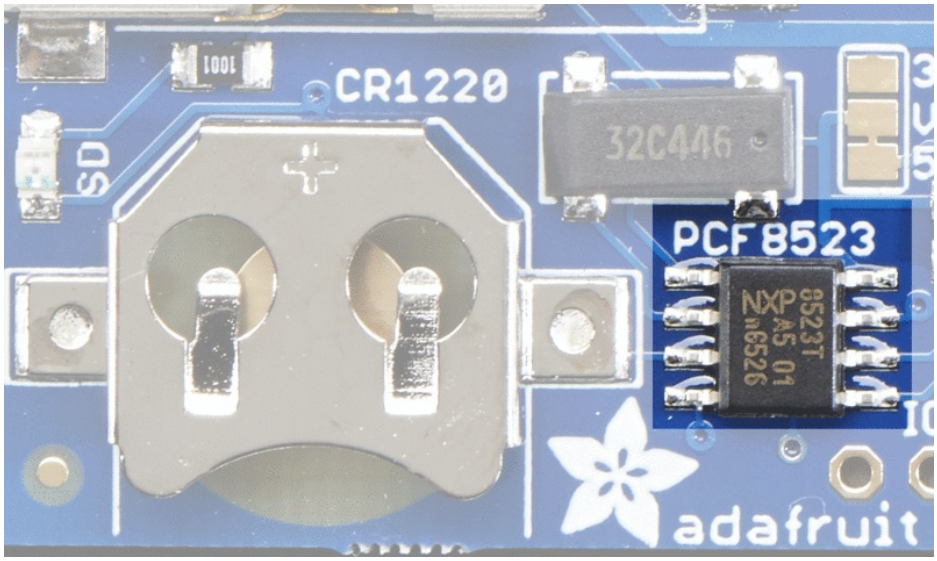
While this sort of basic timekeeping is OK for some projects, a data-logger will need to have **consistent timekeeping that doesnt reset when the Arduino battery dies or is reprogrammed**. Thus, we include a separate RTC! The RTC chip is a specialized chip that just keeps track of time. It can count leap-years and knows how many days are in a month, but it doesn't take care of Daylight Savings Time (because it changes from place to place)



This image shows a computer motherboard with a Real Time Clock called the DS1387 (https://adafru.it/aX0). Theres a lithium battery in there which is why it's so big.

The RTC we'll be using is the *PCF8523 (https://adafru.it/reb)* or the *DS1307* (https://adafru.it/rec).

If you have an Adafruit Datalogger Shield rev B, you will be using the PCF8523 - this RTC is newer and better than the DS1307. Look on your shield to see if you see **PCF8523** written above the chip.



If you have an older Datalogger shield, you will be using the DS1307 - there's no text so you'll just need to remember that if it *doesn't* say PCF8523 it's the DS1307

Battery Backup

As long as it has a coin cell to run it, the RTC will merrily tick along for a long time, even when the Arduino loses power, or is reprogrammed.

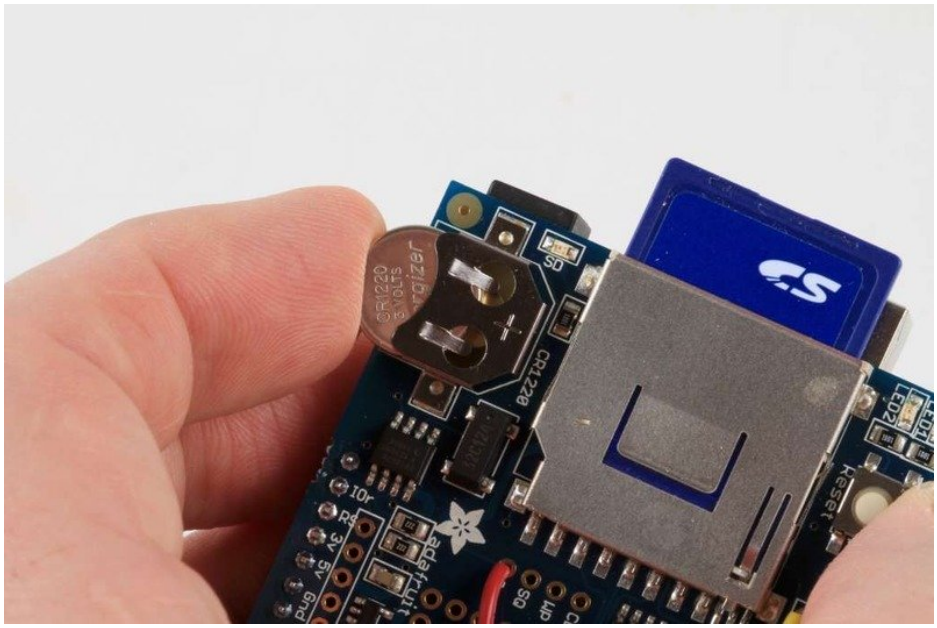
Use any CR1220 3V lithium metal coin cell battery:

CR1220 12mm Diameter - 3V Lithium Coin Cell Battery

$0.95
IN STOCK

Add To Cart

You MUST have a coin cell installed for the RTC to work, if there is no coin cell, it will act strangely and possibly hang the Arduino when you try to use it, so ALWAYS make SURE there's a battery installed, even if it's a dead battery.

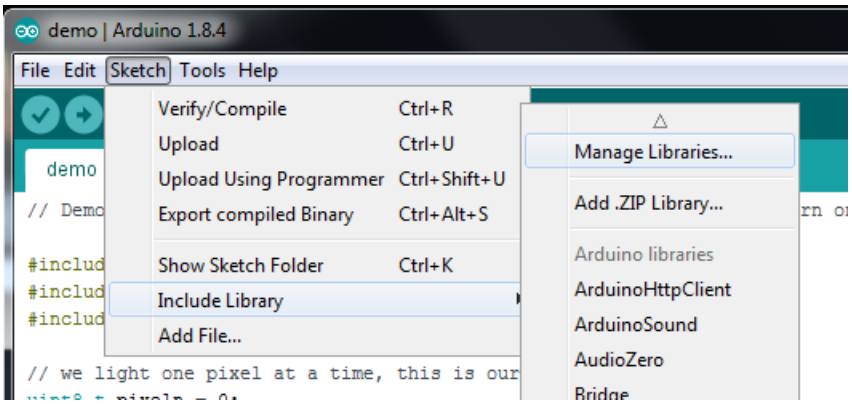


Talking to the RTC

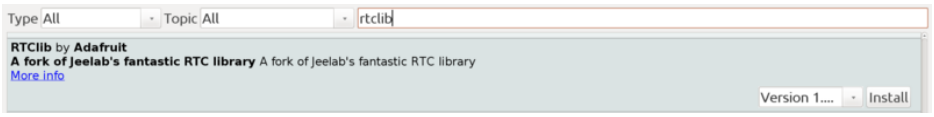
The RTC is an i2c device, which means it uses 2 wires to to communicate. These two wires are used to set the time and retrieve it. On the Arduino UNO, these pins are also wired to the **Analog 4** and **5** pins. This is a bit annoying since of course we want to have up to 6 analog inputs to read data and now we've lost two.

For the RTC library, we'll be using a fork of JeeLab's excellent RTC library, which is available on GitHub (https://adafru.it/c7r). You can install this library via the Arduino Library Manager.

Open up the Arduino Library Manager:



Search for the **RTCLib** library and install it



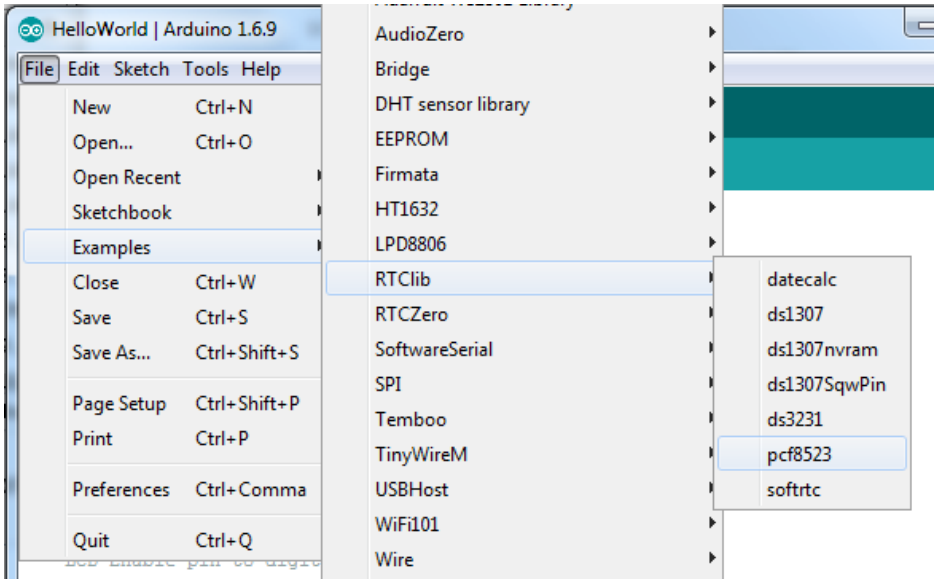
We also have a great tutorial on Arduino library installation at:
http://learn.adafruit.com/adafruit-all-about-arduino-libraries-install-use

First RTC test

The first thing we'll demonstrate is a test sketch that will read the time from the RTC once a second. We'll also show what happens if you remove the battery and replace it since that causes the RTC to halt. So to start, remove the battery from the holder while the Arduino is not powered or plugged into USB. Wait 3 seconds and then replace the battery. This resets the RTC chip. Now load up the matching sketch for your RTC

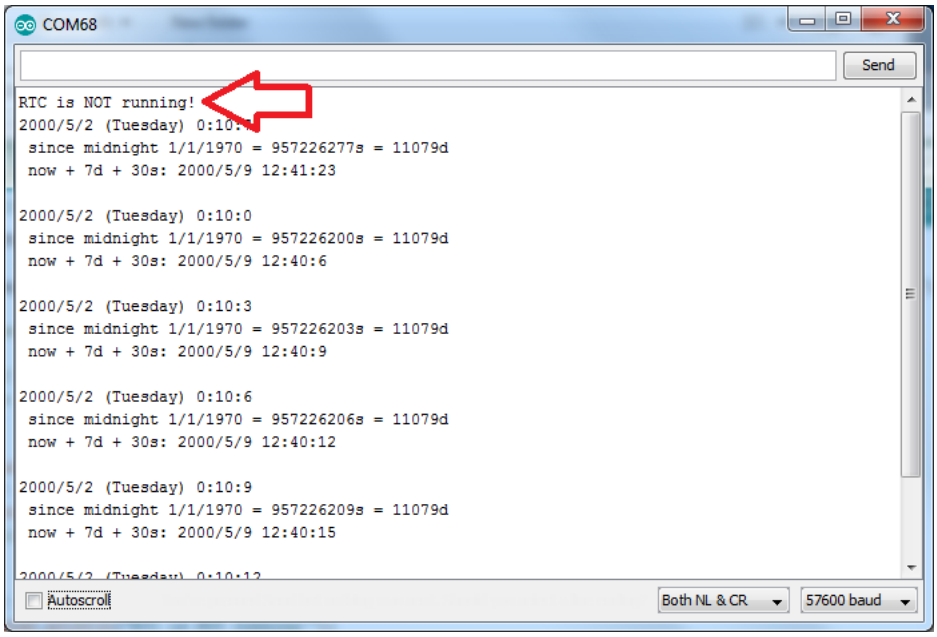
- For the Adafruit Datalogger shield rev B open up **Examples->RTClib->pcf8523**
- For the older Adafruit Dataloggers, use **Examples->RTClib->ds1307**

Upload it to your Arduino with the datalogger shield on!



If you're having problems make sure you are running the right example! PCF8523 and DS1307 RTC chips are not identical so they have separate examples!

Now open up the Serial Console and make sure the baud rate is set correctly at **57600 baud** you should see the following:



Whenever the RTC chip loses all power (including the backup battery) it will reset to an earlier date and report the time as 0:0:0 or similar. The DS1307 won't even count seconds (it's stopped).Whenever you set the time, this will kickstart the clock ticking.

So, basically, the upshot here is that you should never ever remove the battery once you've set the time. You shouldn't have to and the battery holder is very snug so unless the board is crushed, the battery won't 'fall out'

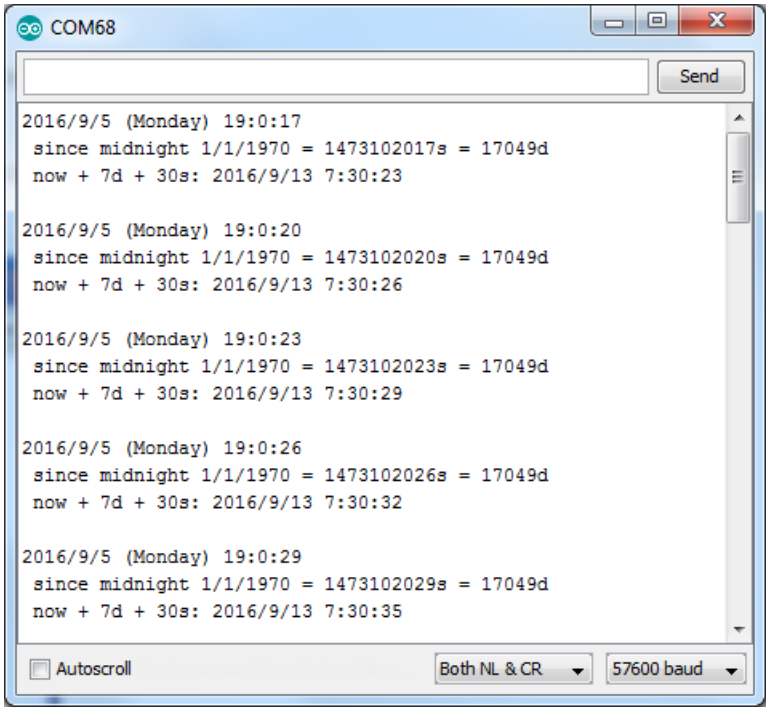
Setting the time

With the same sketch loaded, uncomment the line that starts with **RTC.adjust** like so:

```
if (! rtc.initialized()) {
  Serial.println("RTC is NOT running!");
  // following line sets the RTC to the date & time this sketch was compiled
  rtc.adjust(DateTime(F(__DATE__), F(__TIME__)));
```

This line is very cute, what it does is take the Date and Time according the computer you're using (right when you compile the code) and uses that to program the RTC. If your computer time is not set right you should fix that first. Then you must press the **Upload** button to compile and then immediately upload. If you compile and then upload later, the clock will be off by that amount of time.

Then open up the Serial monitor window to show that the time has been set



From now on, you won't have to ever set the time again: the battery will last 5 or more years

Reading the time

Now that the RTC is merrily ticking away, we'll want to query it for the time. Let's look at the sketch again to see how this is done

```
void loop () {
    DateTime now = rtc.now();

    Serial.print(now.year(), DEC);
    Serial.print('/');
    Serial.print(now.month(), DEC);
    Serial.print('/');
    Serial.print(now.day(), DEC);
    Serial.print(" (");
    Serial.print(daysOfTheWeek[now.dayOfTheWeek()]);
    Serial.print(") ");
    Serial.print(now.hour(), DEC);
    Serial.print(':');
    Serial.print(now.minute(), DEC);
    Serial.print(':');
    Serial.print(now.second(), DEC);
    Serial.println();
```

There's pretty much only one way to get the time using the RTClib, which is to call **now()**, a function that returns a DateTime object that describes the year, month, day, hour, minute and second when you called **now()**.

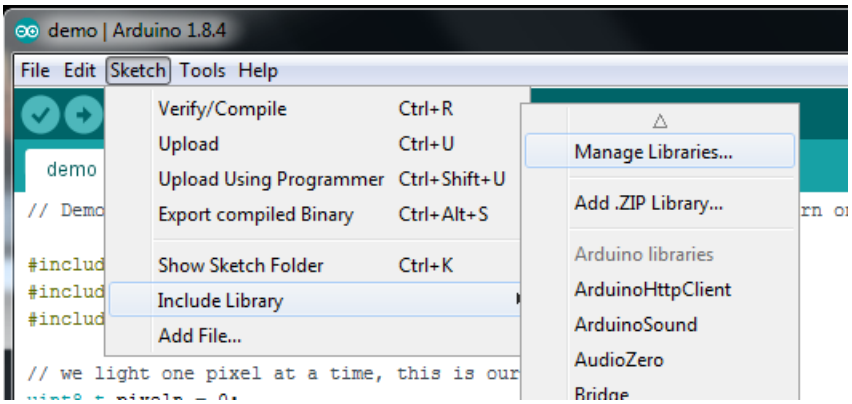
There are some RTC libraries that instead have you call something like **RTC.year()** and **RTC.hour()** to get the current year and hour. However, there's one problem where if you happen to ask for the minute right at **3:14:59** just before the next minute rolls over, and then the second right after the minute rolls over (so at **3:15:00**) you'll see the time as **3:14:00** which is a minute off. If you did it the other way around you could get **3:15:59** - so one minute off in the other direction.

Because this is not an especially unlikely occurance - particularly if you're querying the time pretty often - we take a 'snapshot' of the time from the RTC all at once and then we can pull it apart into **day()** or **second()** as seen above. It's a tiny bit more effort but we think its worth it to avoid mistakes!

We can also get a 'timestamp' out of the DateTime object by calling **unixtime** which counts the number of seconds (not counting leapseconds) since midnight, January 1st 1970

```
    Serial.print(" since 2000 = ");
    Serial.print(now.unixtime());
    Serial.print("s = ");
    Serial.print(now.unixtime() / 86400L);
    Serial.println("d");
```

Since there are 60*60*24 = 86400 seconds in a day, we can easily count days since then as well. This might be useful when you want to keep track of how much time has passed since the last query, making some math a lot easier (like checking if it's been 5 minutes later, just see if **unixtime()** has increased by 300, you dont have to worry about hour changes)



Using the SD Card

The other half of the data logger shield is the SD card. The SD card is how we store long term data. While the Arduino chip has a permanent EEPROM storage, its only a couple hundred bytes - tiny compared to a 2 gig SD card. SD cards are so cheap and easy to get, its an obvious choice for long term storage so we use them for the shield.

The shield kit doesn't come with an SD card but we carry one in the shop that is guaranteed to work (https://adafru.it/aIH). Pretty much any SD card should work but be aware that some cheap cards are 'fakes' and can cause headaches.

4GB Blank SD/MicroSD Memory Card

OUT OF STOCK

Out Of Stock

You'll also need a way to read and write from the SD card. Sometimes you can use your camera and MP3 player - when its plugged in you will be able to see it as a disk. Or you may need an SD card reader (http://adafru.it/939). The shield **doesnt** have the ability to display the SD card as a 'hard disk' like some MP3 players or games, the Arduino does not have the hardware for that, so you will need an external reader!

USB MicroSD Card Reader/Writer - microSD / microSDHC / microSDXC

$5.95
IN STOCK

Add To Cart

Formatting under Windows/Mac

If you bought an SD card, chances are it's already pre-formatted with a FAT filesystem. However you may have problems with how the factory formats the card, or if it's an old card it needs to be reformatted. The Arduino SD library we use supports both **FAT16** and **FAT32** filesystems. If you have a very small SD card, say 8-32 Megabytes you might find it is formatted **FAT12** which isnt supported. You'll have to reformat these card. Either way, its **always** good idea to format the card before using, even if its new! Note that formatting will erase the card so save anything you want first

We strongly recommend you use the official SD card formatter utility - written by the SD association it solves many problems that come with bad formatting!

The official SD formatter is available from https://www.sdcard.org/downloads/formatter_4/ (https://adafru.it/cfL)

Download it and run it on your computer, there's also a manual linked from that page for use

https://adafru.it/cfL

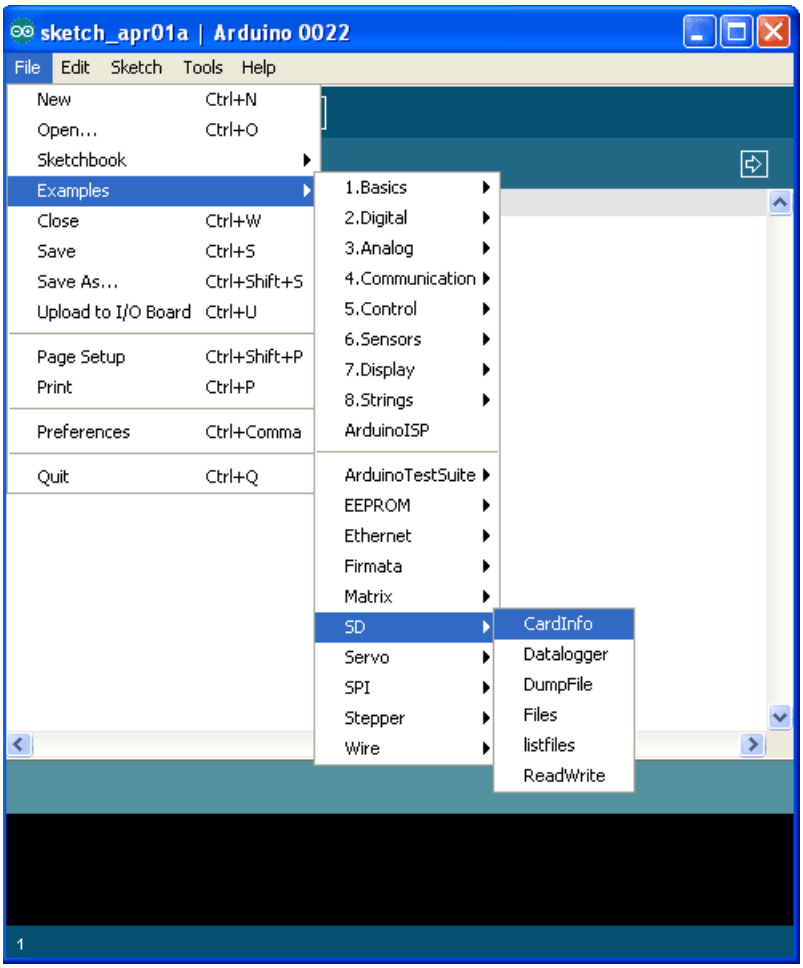
https://adafru.it/cfL

Get Card Info

The Arduino SD Card library has a built in example that will help you test the shield and your connections

> If you have an older Datalogging shield without the SPI header connection **and** you are using a **Leonardo, Mega** or anything other than an UNO, you'll need to install a special version of the SD library (https://adafru.it/ref)

Open the file **CardInfo** example sketch in the **SD** library:



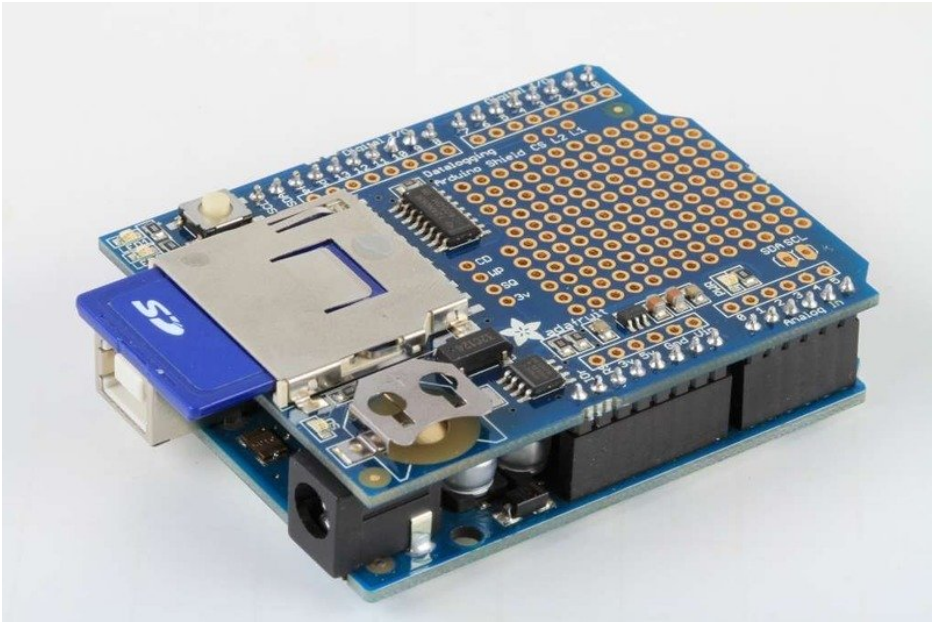
This sketch will not write any data to the card, just tell you if it managed to recognize it, and some information about it.

This can be **very** useful when trying to figure out whether an SD card is supported. Before trying out a new card, please try out this sketch!

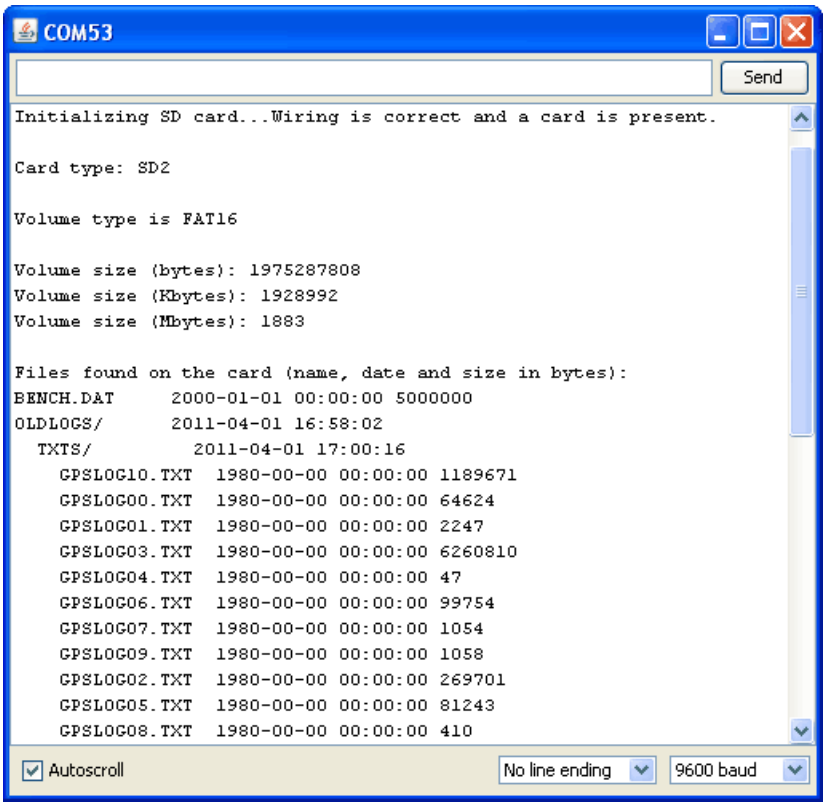
Go to the beginning of the sketch and make sure that the **chipSelect** line is correct, for the datalogger shield we 're using digital pin 10 so change it to 10!

If you have the pre-rev B version of the Datalogger Shield, and you are using a Mega or Leonardo check here for now to adjust the pin setup (https://adafru.it/ref)

OK, now insert the SD card into the Arduino and upload the sketch

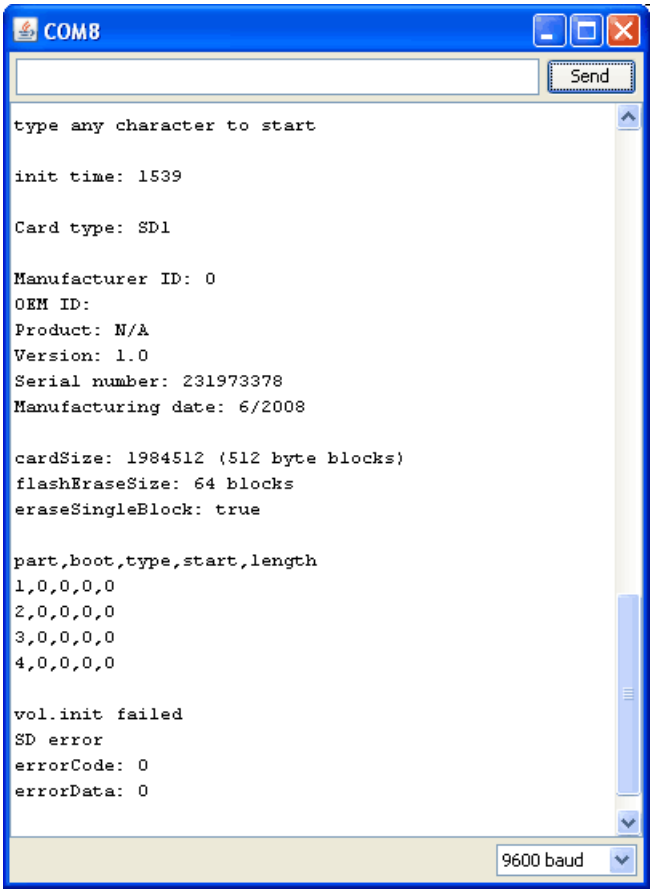


Open up the Serial Monitor and type in a character into the text box (& hit send) when prompted. You'll probably get something like the following:



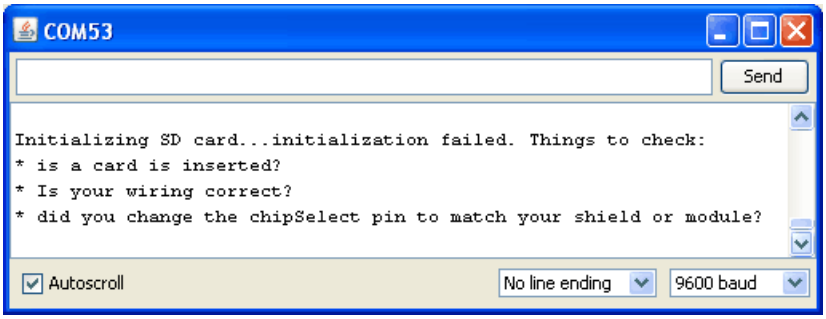
Its mostly gibberish, but its useful to see the **Volume type is FAT16** part as well as the size of the card (about 2 GB which is what it should be) etc.

If you have a bad card, which seems to happen more with ripoff version of good brands, you might see:



The card mostly responded, but the data is all bad. Note that the **Product ID** is **"N/A"** and there is no **Manufacturer ID** or **OEM ID**. This card returned some SD errors. Its basically a bad scene, I only keep this card around to use as an example of a bad card! If you get something like this (where there is a response but its corrupted) you should toss the card

Finally, try taking out the SD card and running the sketch again, you'll get the following,



It couldn't even initialize the SD card. This can also happen if there's a soldering error or if the card is *really* damaged

If you're having SD card problems, we suggest using the SD formatter mentioned above first to make sure the card is clean and ready to use!

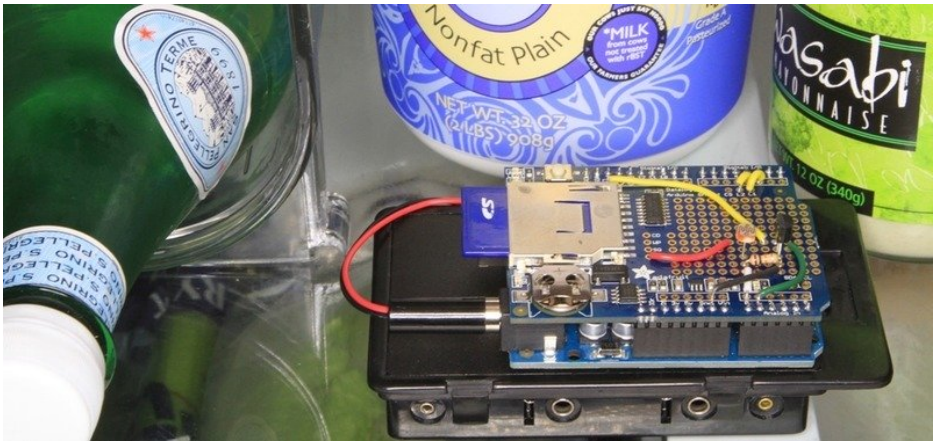
Light and Temperature Logger

Introduction

OK now that we have introduced both the RTC and the SD card and verified that they're working, we can move onto logging!

We'll use a pretty good & detailed demonstration to show off the capabilities of this most awesome data logging shield: We'll log both temperature and relative light levels to determine:

1. How much does the temperature in a fridge vary as the compressor turns on and off?
2. Does keeping the door open cause a big temperature drop? How long does it take for it to cool down?
3. Does the light inside *really* turn off when the door is closed?



Build It!

Items you'll need:

- Arduino (of course!) a Atmega328 type is best (http://adafru.it/50)- we always recommend going with an official 'classic' Arduino such as the Uno.
- Adafruit data logger shield (http://adafru.it/1141) - assembled
- SD card formatted for FAT (http://adafru.it/102) and tested using our example sketch (https://adafru.it/cIN)
- CdS photocell (http://adafru.it/161) and a matching 10K pulldown resistor
- Temperature sensor with analog out, such as TMP36 (http://adafru.it/165)
- Battery pack such as a 6-AA 'brick' and a 2.1mm DC jack. (http://adafru.it/248)
- **or** you can use a 9V clip for a power supply(http://adafru.it/80) but a 9V powered logger will last only a couple hours so we suggest 6xAA's
- Some 22 AWG wire (https://adafru.it/c79), soldering iron, solder (https://adafru.it/doU), etc.

You can get most everything in that list in a discounted pack in the Adafruit shop!(http://adafru.it/249)

The sensors

We'll use two basic sensors to log data, a CdS photocell to track light (http://adafru.it/161) (this will tell us when the door has been opened) and a semiconductor temperature sensor to log the ambient fridge temperature. (http://adafru.it/165)

We have two great tutorials for these sensors on our site, if you haven't used them before or need some refreshment, please read them now!

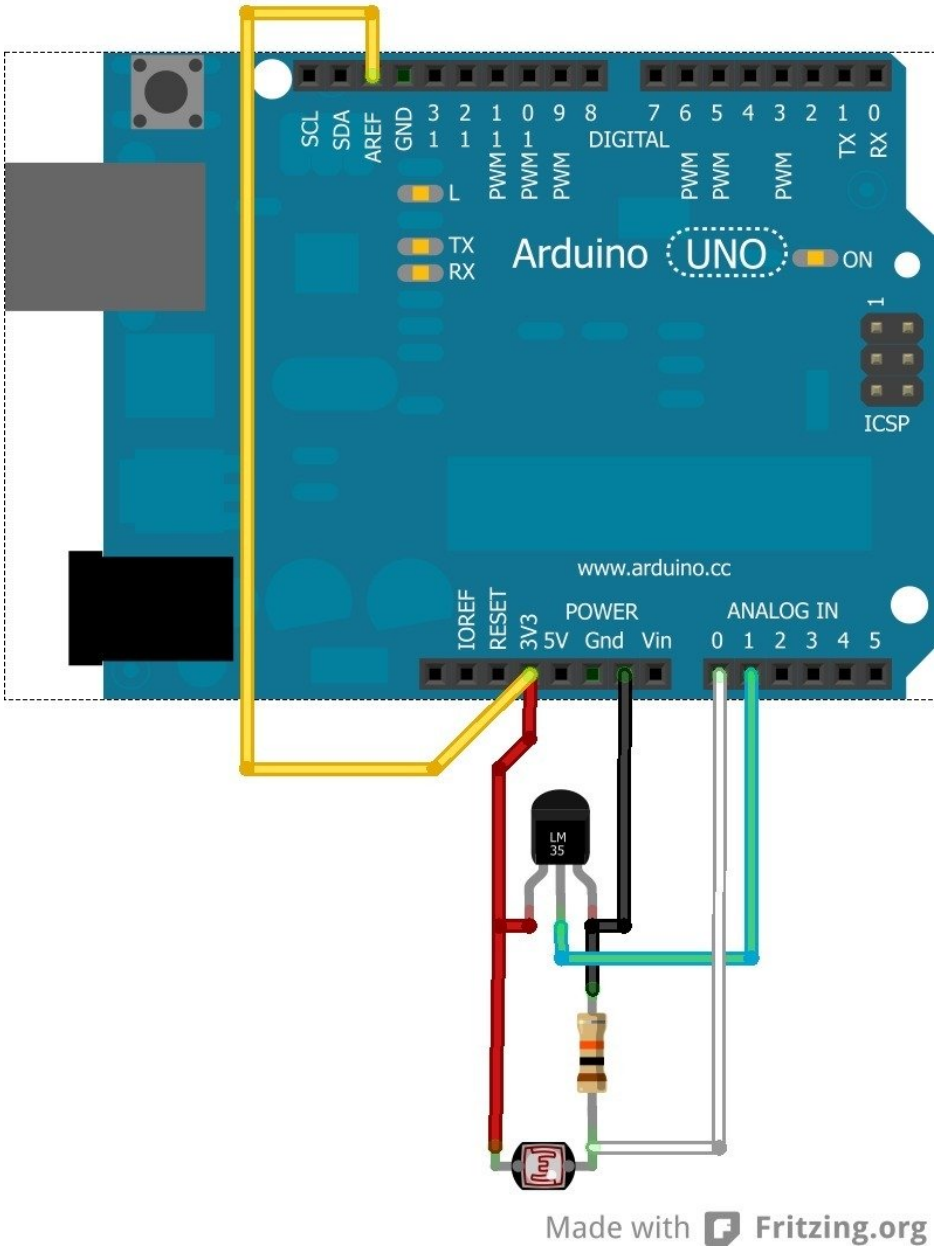
https://adafru.it/reg

https://adafru.it/reg

https://adafru.it/reh

We will wire the sensors as shown in the diagram below.

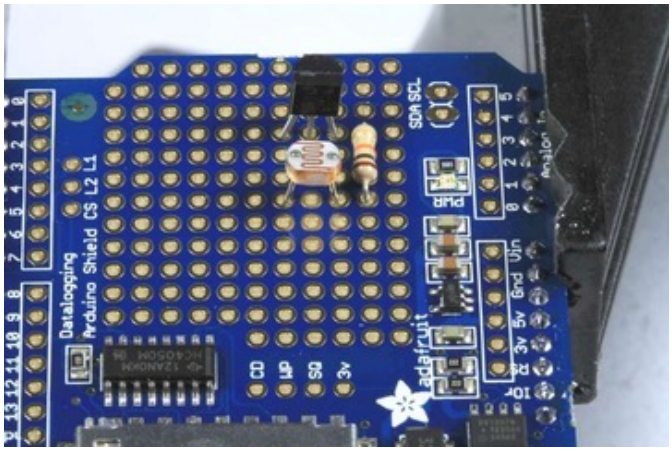
Note that we connect **ARef,** the power pin of the temp sensor, and the light sensor to **3.3V** *not to 5.0V* - we do this because the 5V line is very noisy and the 3.3V regulator is better filtered. In the actual board we used the 3.3V line from the datalogger's regulator, see the images below - in theory its the same as the one off of the Arduino but we trust ours more.



Wiring it up

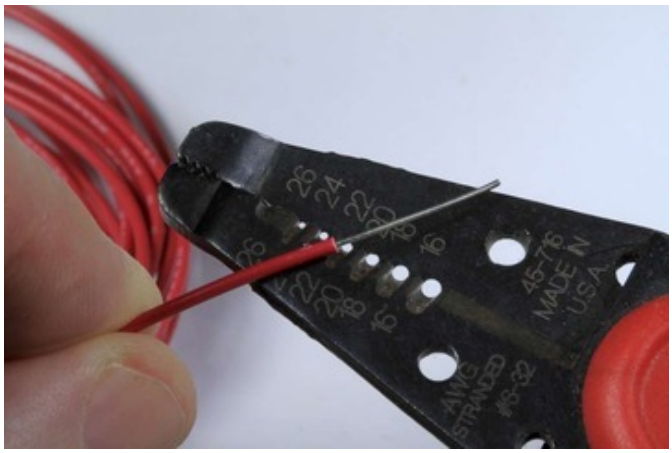
The prototyping area on the board is a simple array of holes with soldering pads. The steps below show how we built this circuit and illustrate some some basic circuit prototyping techniques. For clarity, we will use the same color wire as

shown in the circuit diagram above:



Position the sensors

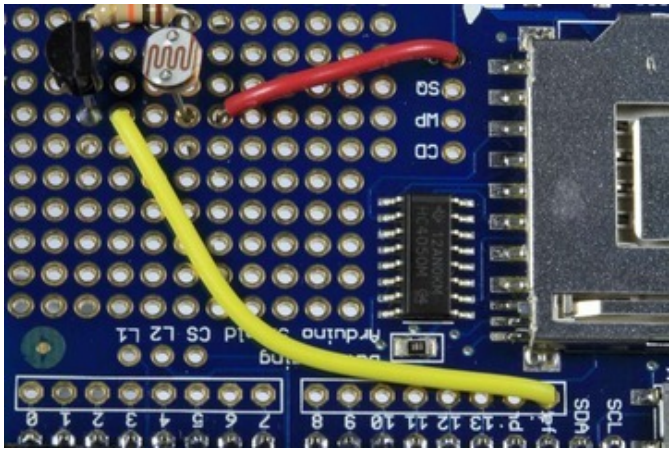
The sensors could go anywhere on the prototyping area, but we chose this arrangement to simplify connections between the components later on.



Prepare some jumpers

Measure a piece of wire (red) long enough to reach from the 3v breakout hole to 1/2" past the temperature sensor. Strip about 3/4" from one end, and about 1/4" from the other.

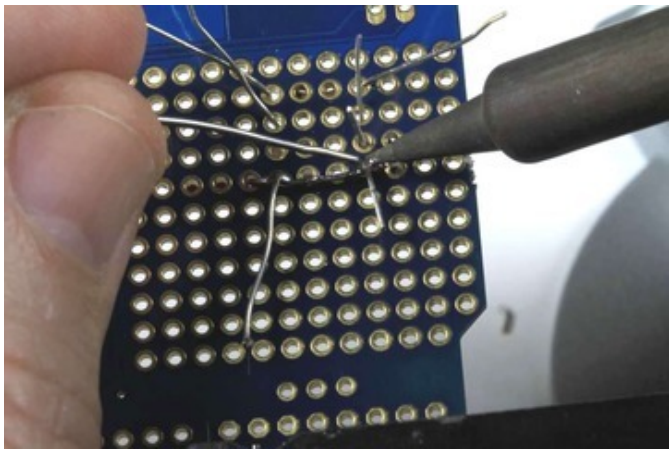
Measure another one (yellow) long enough to reach from the AREF pin to the hole between the two sensors. Strip 1/2" from one end and 1/4" from the other.



Install the Jumpers

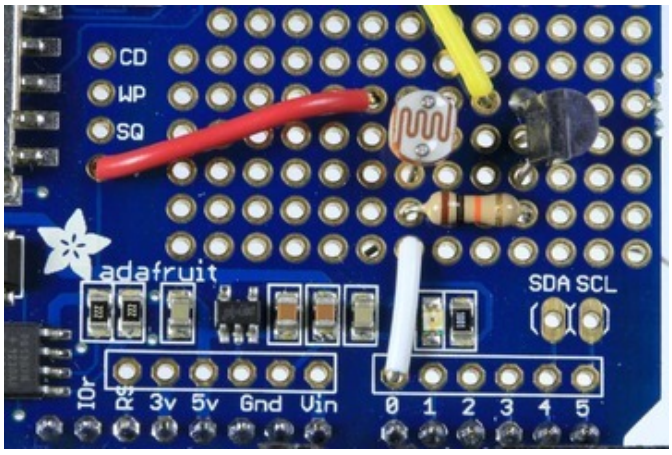
Place the jumpers as shown, with the long stripped ends nearest the sensors.

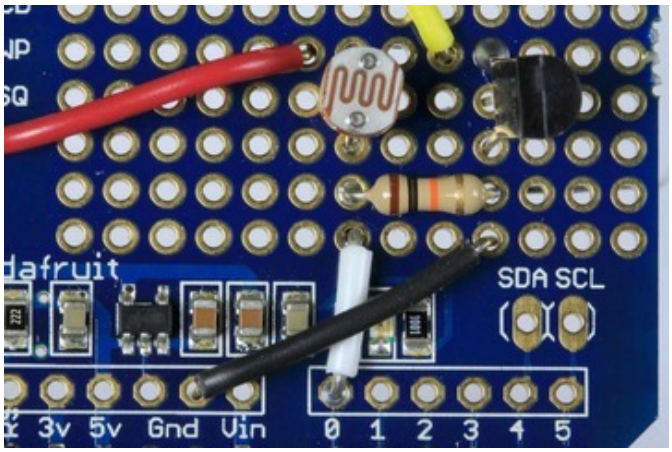
Since there are no signal traces between the holes in the prototyping area, we will use the long stripped ends to join the legs of the components on the board.

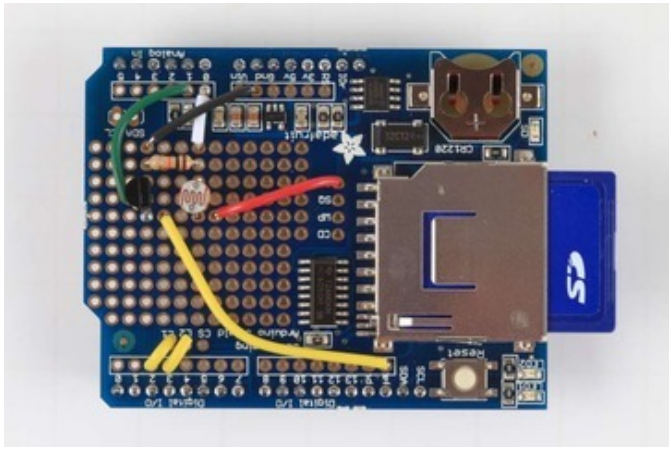


Make the connections

- Solder the first jumper (red) to the 3v hole.
- Bend the stripped end of the wire so it rests next to the legs of the light sensor, the temperature sensor and the end of the AREF jumper.
- Fold the sensor legs and AREF jumper legs over the 3v jumper and solder to make the connection.





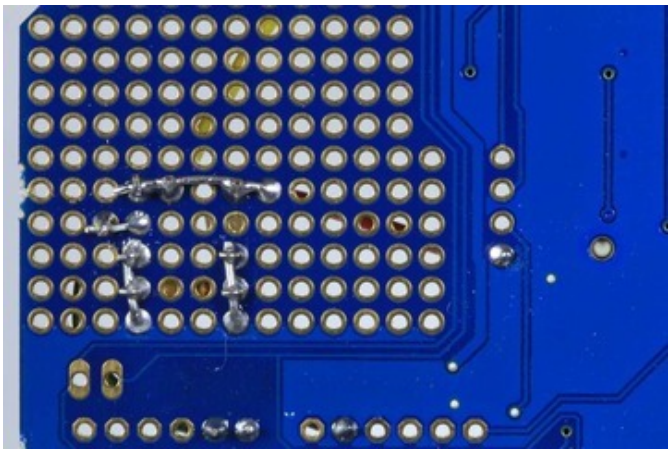


Add more jumpers for the Sensors

- From Analog Pin 0 to the hole near the light sensor and resistor. (white)
- From GND to the hole next to the other end of the resistor (black)
- From the Analog pin 1 to the hole next to the center pin of the temperature sensor (green)

And also for the LEDs

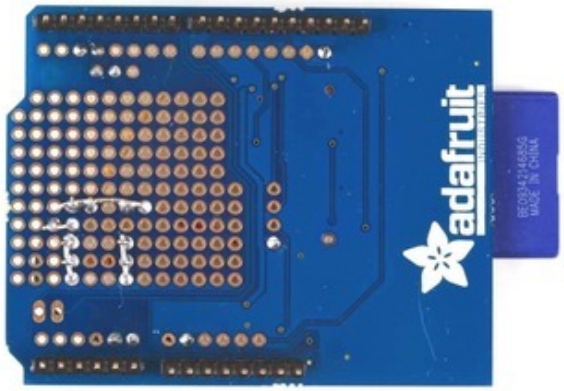
- From L1 to Digital Pin 2 (yellow)
- From L2 to Digital Pin 3 (yellow)

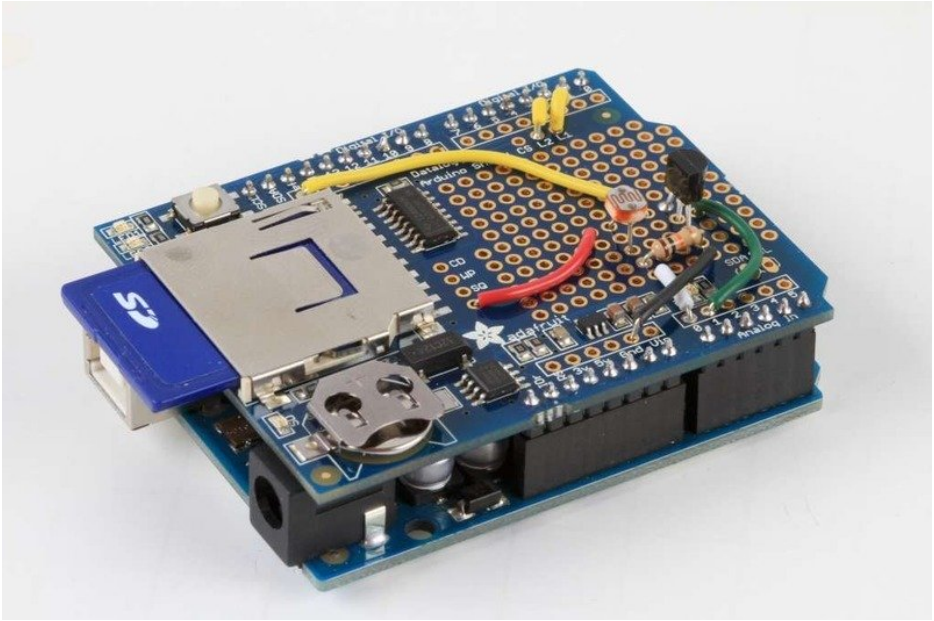


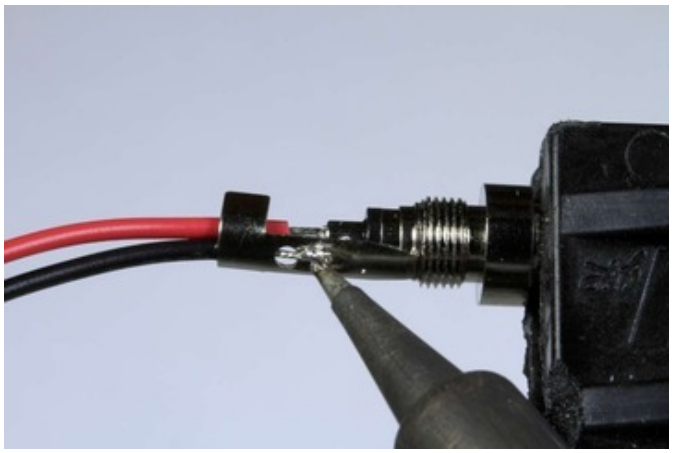
Solder and trim all connections

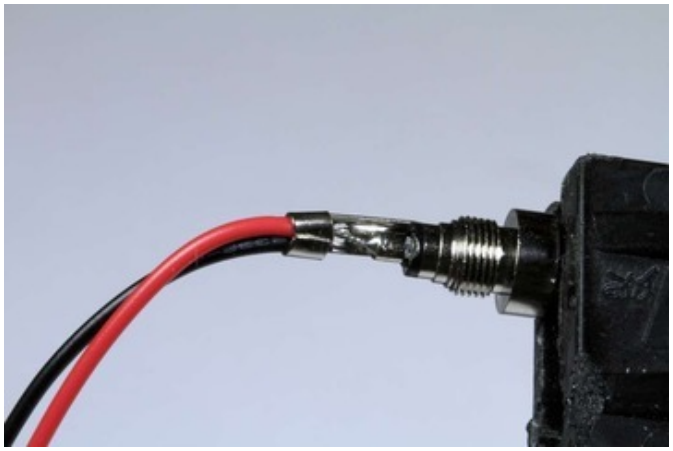
Using the same technique of folding the component legs over the jumper - make all connections as shown in the wiring diagram.

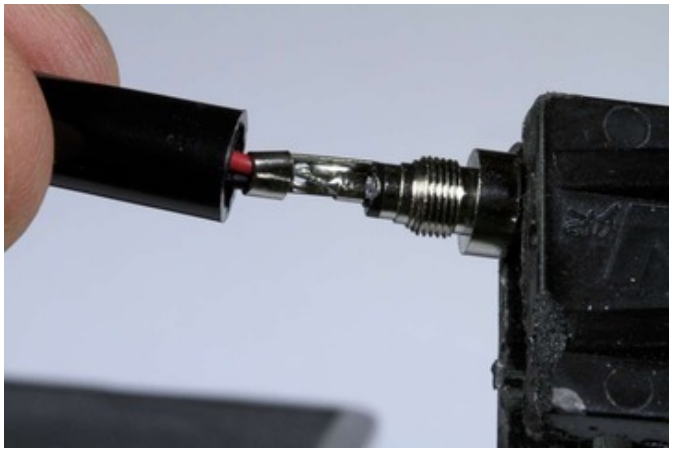
Make sure that all connections are soldered. Also solder wires and component legs to the board where they pass through the holes.











Prepare the Battery Pack

- Place the black plastic ferrule from the connector over the battery pack wires.
- Solder the red wire from the battery pack to the center pin
- Solder the the black wire to the outer barrel.
- Crimp to hold the wires securely
- Screw the black plastic ferrule on to cover the solder joints.

Now your Light Temp Logger is wired and ready for testing!

Use It!

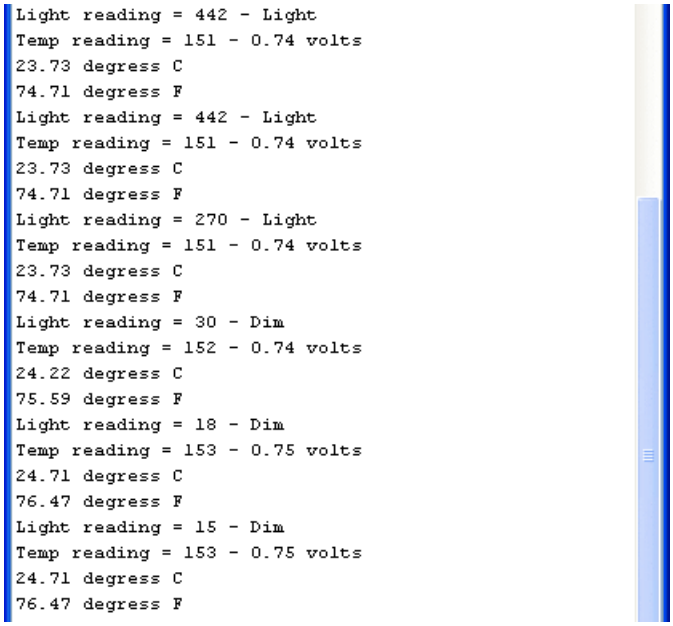
Sensor test

We'll now test the sensors, using this sketch which is a bit of a mashup of the two examples in our tutorials (https://adafru.it/c7d)

```
#include <SPI.h>
#include <SD.h>

/* Sensor test sketch
  for more information see http://www.ladyada.net/make/logshield/lighttemp.html
  */

#define aref_voltage 3.3         // we tie 3.3V to ARef and measure it with a multimeter!

int photocellPin = 0;     // the cell and 10K pulldown are connected to a0
int photocellReading;     // the analog reading from the analog resistor divider

//TMP36 Pin Variables
int tempPin = 1;        //the analog pin the TMP36's Vout (sense) pin is connected to
                        //the resolution is 10 mV / degree centigrade with a
                        //500 mV offset to allow for negative temperatures
int tempReading;        // the analog reading from the sensor

void setup(void) {
  // We'll send debugging information via the Serial monitor
  Serial.begin(9600);

  // If you want to set the aref to something other than 5v
  analogReference(EXTERNAL);
}


void loop(void) {
  photocellReading = analogRead(photocellPin);

  Serial.print("Light reading = ");
  Serial.print(photocellReading);     // the raw analog reading

  // We'll have a few threshholds, qualitatively determined
  if (photocellReading < 10) {
    Serial.println(" - Dark");
  } else if (photocellReading < 200) {
    Serial.println(" - Dim");
  } else if (photocellReading < 500) {
    Serial.println(" - Light");
  } else if (photocellReading < 800) {
    Serial.println(" - Bright");
  } else {
    Serial.println(" - Very bright");
  }

  tempReading = analogRead(tempPin);

  Serial.print("Temp reading = ");
  Serial.print(tempReading);     // the raw analog reading
```

```
  // converting that reading to voltage, which is based off the reference voltage
  float voltage = tempReading * aref_voltage / 1024;

  // print out the voltage
  Serial.print(" - ");
  Serial.print(voltage); Serial.println(" volts");

  // now print out the temperature
  float temperatureC = (voltage - 0.5) * 100 ;  //converting from 10 mv per degree wit 500 mV offset
                                                //to degrees ((volatge - 500mV) times 100)
  Serial.print(temperatureC); Serial.println(" degrees C");

  // now convert to Fahrenheight
  float temperatureF = (temperatureC * 9 / 5) + 32;
  Serial.print(temperatureF); Serial.println(" degrees F");

  delay(1000);
}
```

OK upload this sketch and check the Serial monitor again



In my workroom, I got about 24 degrees C and a 'light measurement' of about 400 - remember that while the temperature sensor gives an 'absolute' reading in C or F, the light sensor is not precise and can only really give rough readings.

Once you've verified that the sensors are wired up correctly & running its time to get to the logging!

Logging sketch

Download the light and temperature logging sketch from GitHub (https://adafru.it/c7e). Insert the SD card.

Look at the top of the sketch for this section and uncomment whichever line is relevant. Check the RTC page for details if you're not sure which one you have. (https://adafru.it/rei)

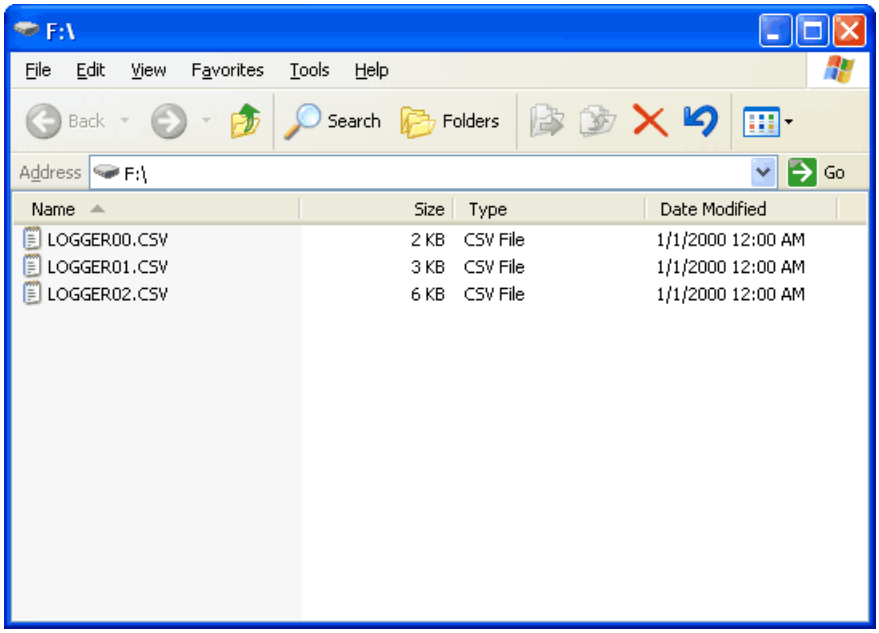
```
/************** if you have a DS1307 uncomment this line **************/

//RTC_DS1307 RTC; // define the Real Time Clock object

/************** if you have a PCF8523 uncomment this line **************/

//RTC_PCF8523 RTC; // define the Real Time Clock object

/**********************************************************************/
```

Upload the sketch to your Arduino. We'll now test it out while still 'tethered' to the computer

While the Arduno is still connected, blinking and powered, place your hand over the photocell for a few seconds, then shine a flashlight on it. You should also squeeze the temp sensor with your fingers to heat it up

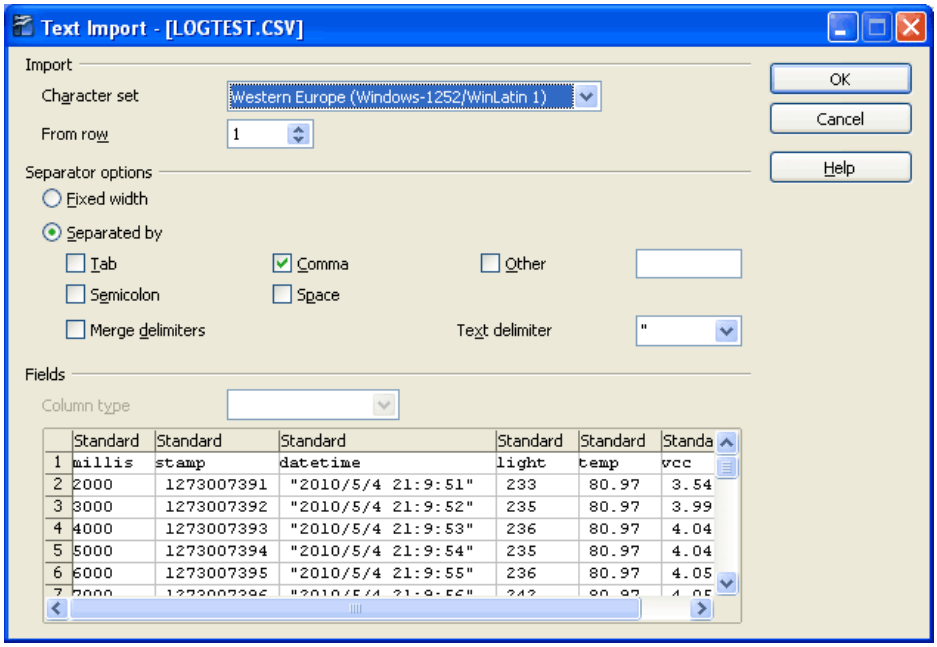
Plotting with a spreadsheet

When you're ready to check out the data, unplug the Arduino and put the SD card into your computer's card reader. You'll see a at least one and perhaps a couple files, one for each time the logger ended up running

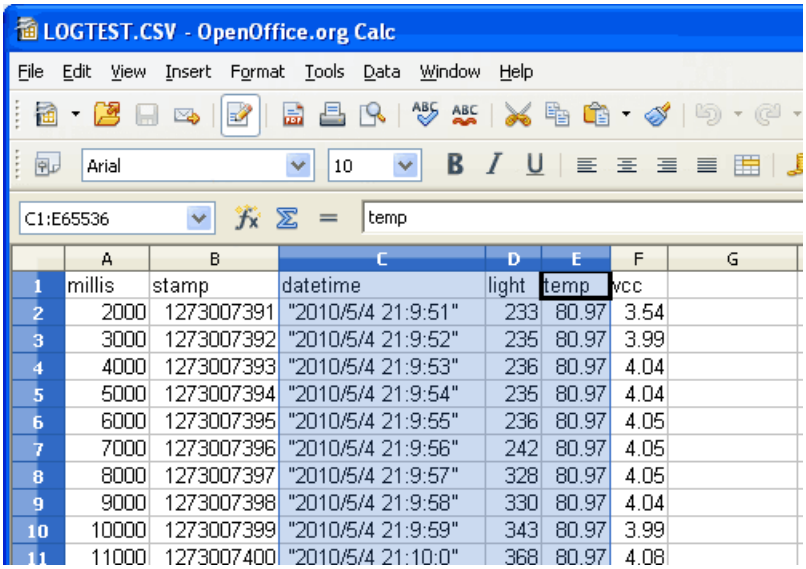


We'll open the most recent one. If you want to use the same logfile used in the graphing demos, click here to download it (https://adafru.it/cny).

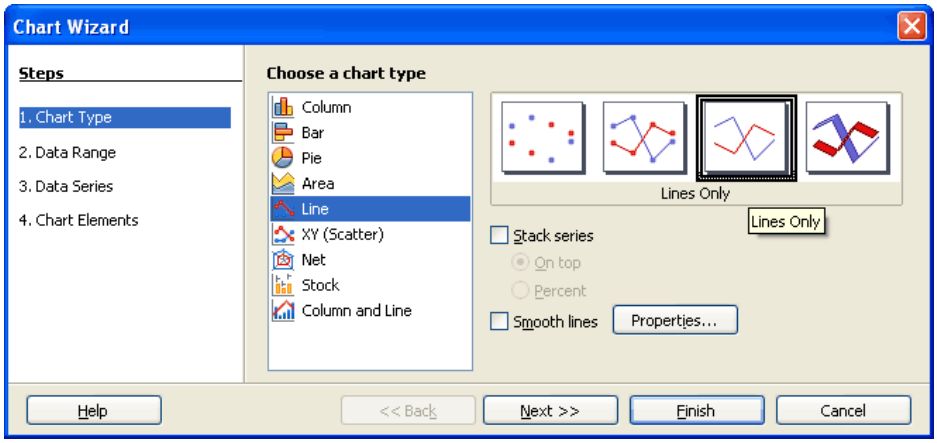
The quickest way to look at the data is using something like OpenOffice or Excel, where you can open the .csv file and have it imported directly into the spreadsheet



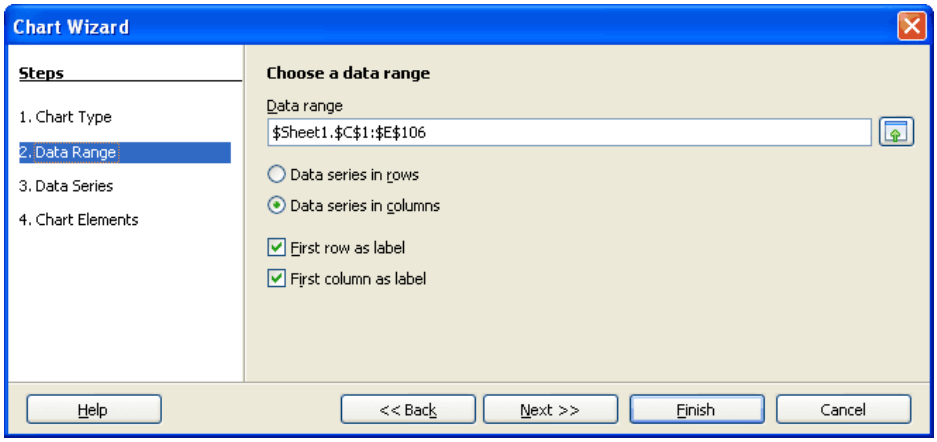
You can then perform some graphing by selecting the columns of data

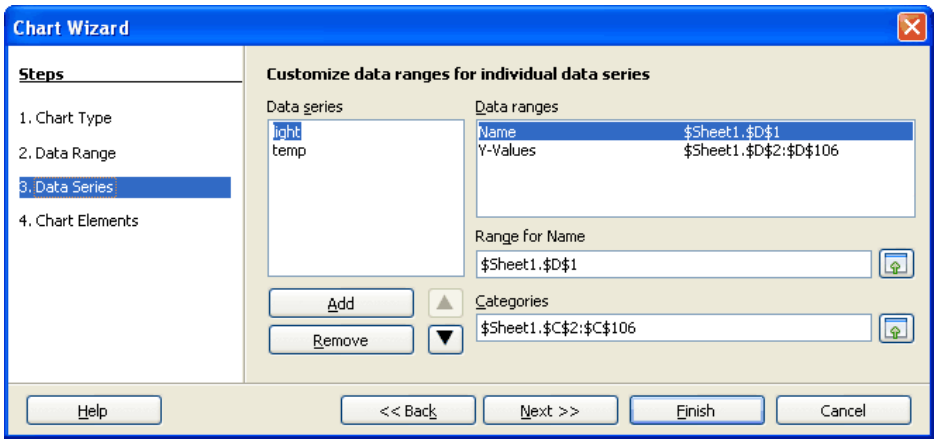


Clicking the **Chart** button and using Lines (we think they are the best for such graphs)

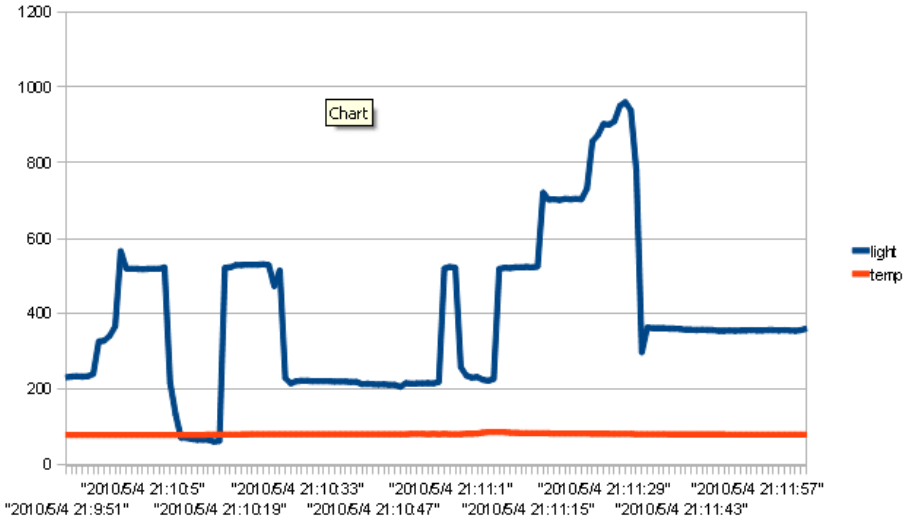


Setting the **First Column as label**



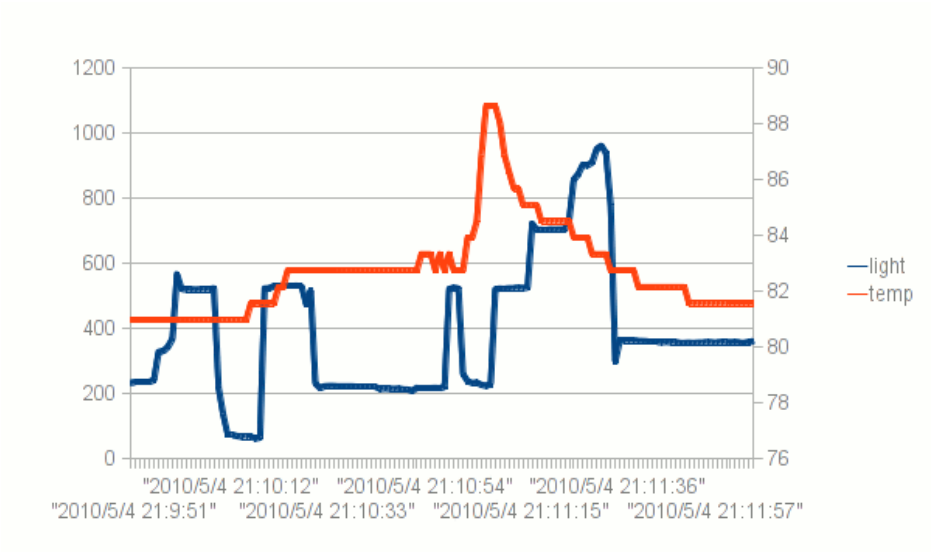


Which will generate this graph

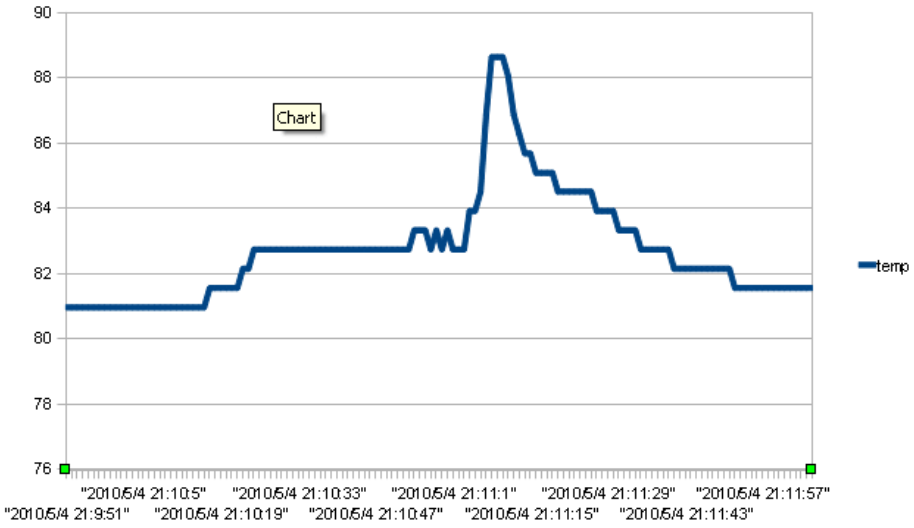


You can see pretty clearly how I shaded the sensor and then shone a flashlight on it.

You can make the graph display both with different axes (since the change in temperature is a different set of units. Select the temp line (red), right-click and choose **Format Data Series**. In the **Options** tab, **Align data series to Secondary Y-axis**.



Or you can make another graph with only the **temp** data



Now you can see clearly how I warmed up the sensor by holding it between my fingers

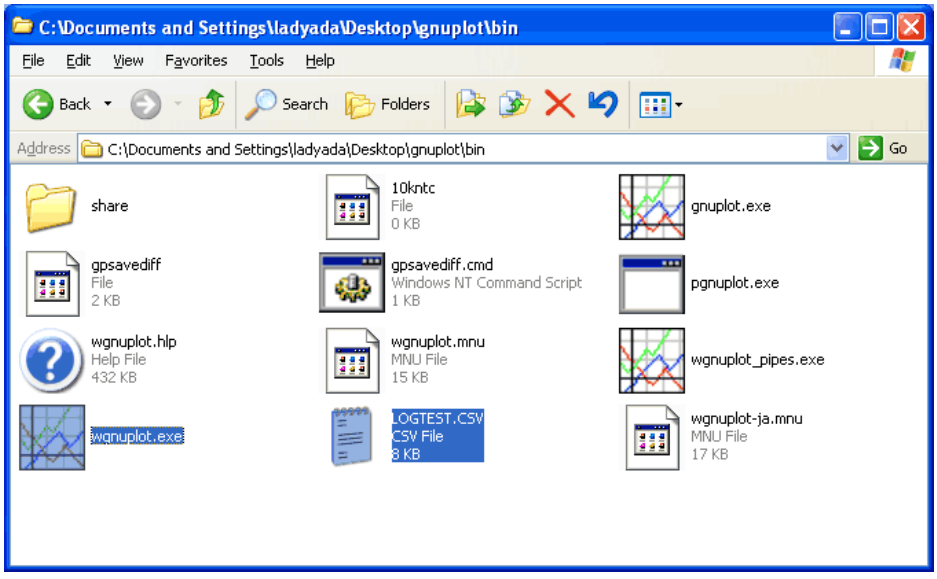
Using Gnuplot

Gnuplot is an free (but not open source?), ultra-powerful plotting program. Its also a real pain to use! But if you can't afford a professional math/plotting package such as Mathematica or Matlab, Gnuplot can do a lot!

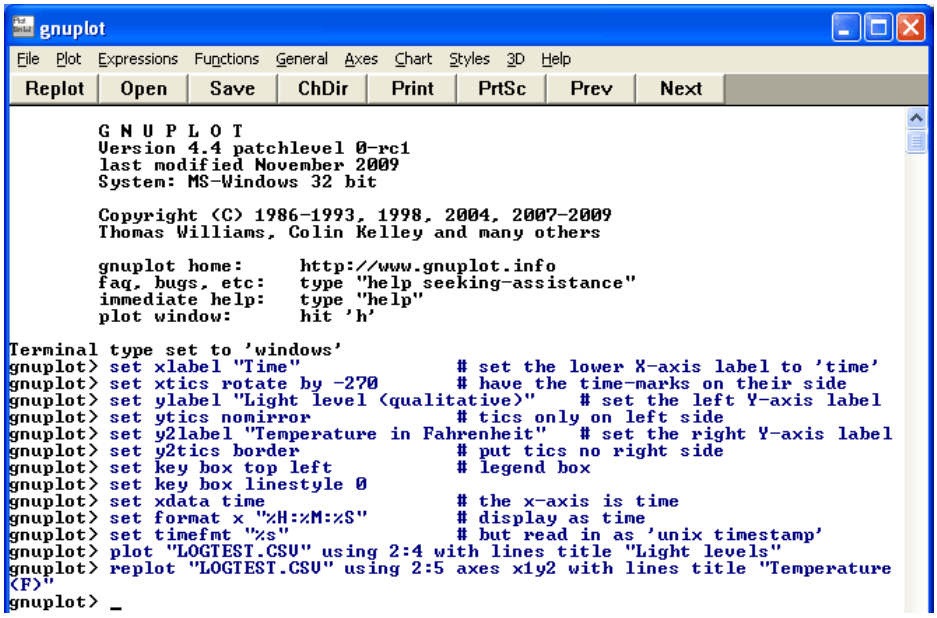
We're not good enough to provide a full tutorial on gnuplot, here are a few links we found handy. Google will definitely help you find even more tutorials and links. Mucking about is the best teacher, too!

- http://www.cs.hmc.edu/~vrable/gnuplot/using-gnuplot.html (https://adafru.it/c7i)
- http://www.duke.edu/~hpgavin/gnuplot.html (https://adafru.it/c7k)
- http://www.ibm.com/developerworks/library/l-gnuplot/ (https://adafru.it/c7m)

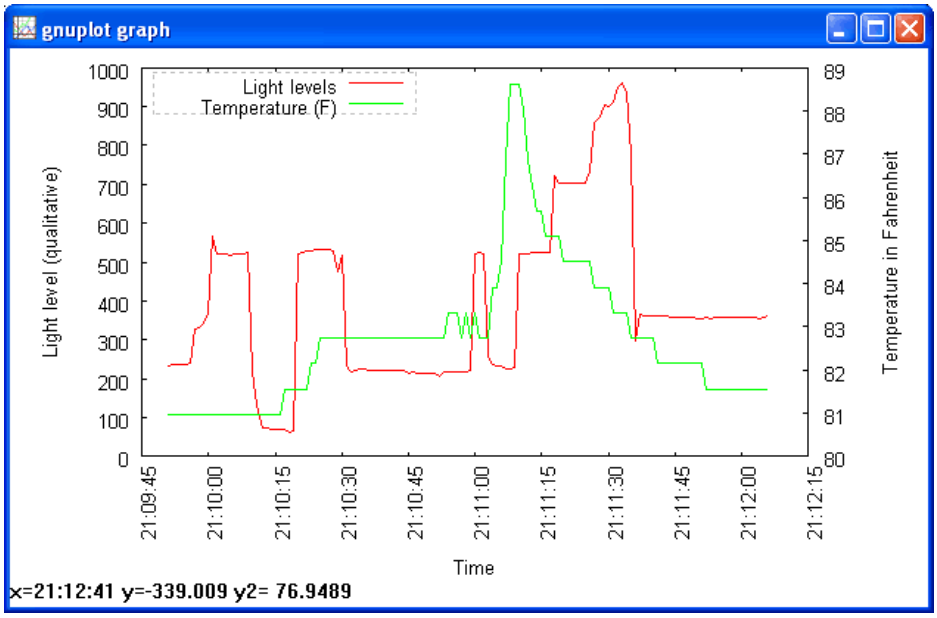
We found the following commands executed in order will generate a nice graph of this data, be sure to put LOGTEST.CSV in the same directory as **wgnuplot.exe** (or if you know how to reference directories, you can put it elsewhere)



```
set xlabel "Time"             # set the lower X-axis label to 'time'

set xtics rotate by -270      # have the time-marks on their side


set ylabel "Light level (qualitative)"    # set the left Y-axis label

set ytics nomirror            # tics only on left side

set y2label "Temperature in Fahrenheit"   # set the right Y-axis label
set y2tics border             # put tics no right side

set key box top left          # legend box
set key box linestyle 0

set xdata time                # the x-axis is time
set format x "%H:%M:%S"       # display as time
set timefmt "%s"              # but read in as 'unix timestamp'

plot "LOGTEST.CSV" using 2:4 with lines title "Light levels"
replot "LOGTEST.CSV" using 2:5 axes x1y2 with lines title "Temperature (F)"
```



Which makes this:



Note the cool double-sided y-axis scales! You can zoom in on stuff pretty easily too.

Other plotters

Our friend John also suggests Live-Graph as a free plotting program (https://adafru.it/c7o) (https://adafru.it/c7o) - we haven't tried it but its worth looking at if you need to do a lot of plotting!

Portable logging

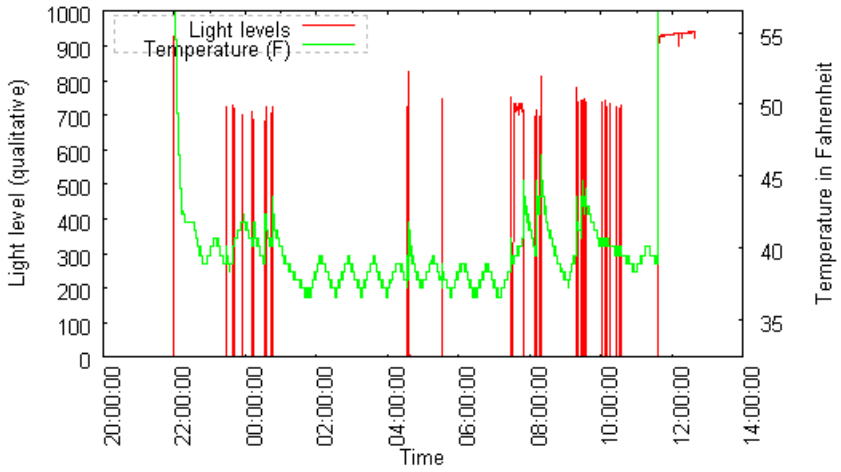
Of course, having a datalogger thats chained to a desktop computer isn't that handy. We can make a portable logger with the addition of a battery pack. The cheapest way to get a good amount of power is to use 6 AA batteries. I made one here with rechargables and a 6xAA battery holder (http://adafru.it/248). It ran the Arduino logging once a second for 18.5 hours. If you use alkalines you could easily get 24 hours or more.

Fridge logging

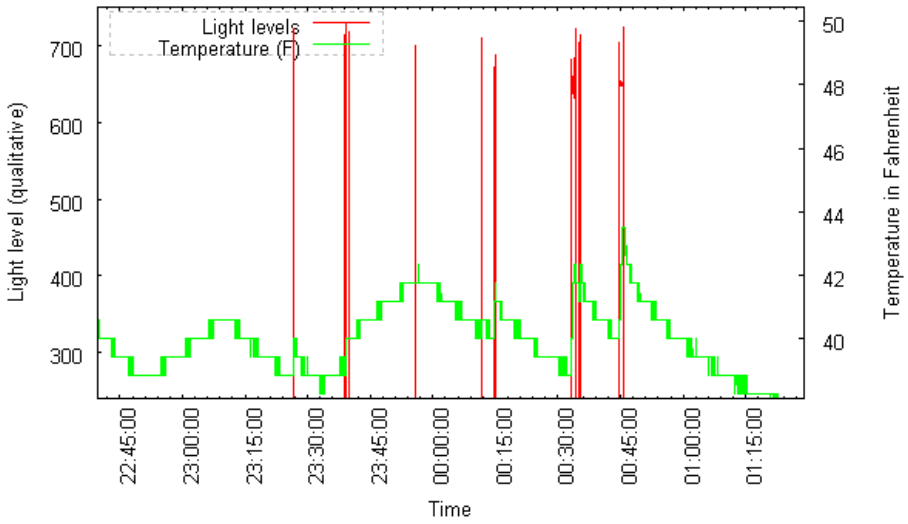
With my portable logger ready, its time to do some Fridge Loggin'! Both were placed in the fridge, in the center of the middle shelf.

I placed it in around 10PM and then removed it around noon the next day. If you don't have a fridge handy, you can grab the data from this zip file and use that (https://adafru.it/cnz).

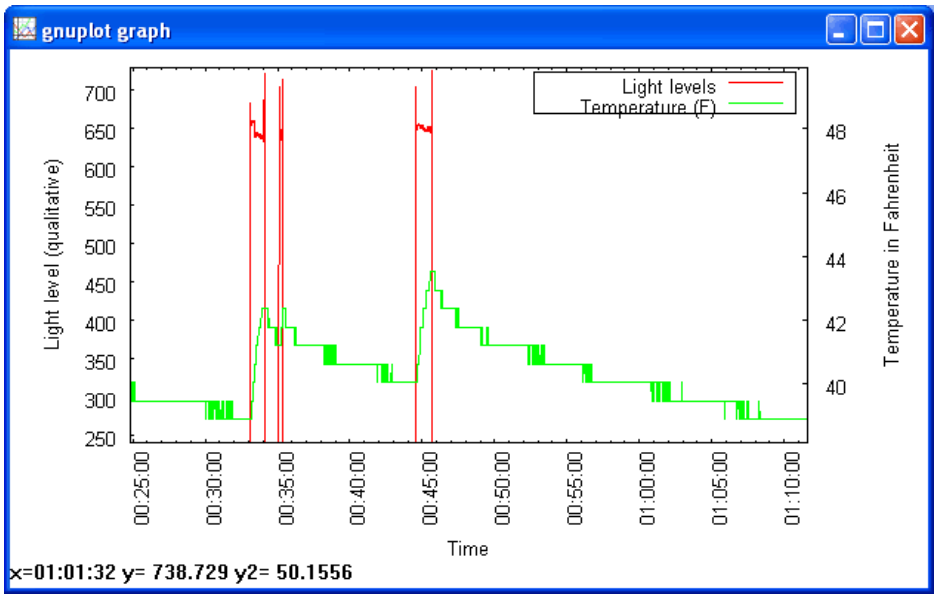
Here is the logged data:



You can see in the middle and end the temp and light levels are very high because the logger was outside the fridge. The green line is the temperature so you can see the temperature slowly rising and then the compressor kicking in every half hour or so. The red lines indicate when the door was opened. This night was a more insominac one than normal!



Zooming into the plot at about 12:40AM, we can see how the temperature climbs whenever the door is open, even in a few seconds it can climb 4 degrees very quickly!



Conclusion!

OK that was a detailed project but its a good one to test your datalogging abilities, especially since its harder to fix bugs in the field. In general, we suggest trying other sensors and testing them at home if possible. Its also a good idea to log more data than you need, and use a software program to filter anything you dont need. For example, we dont use the VCC log but if you're having strange sensor behavior, it may give you clues if your battery life is affecting it.

Code Walkthrough

Introduction

This is a walkthrough of the Light and Temperature Logging sketch. Its long and detailed so we put it here for your perusal. We strongly suggest reading through it, the code is very versatile and our text descriptions should make it clear why everything is there!

Download the complete file here (https://adafru.it/c7e):

Includes and Defines

```
#include "SD.h"
#include <Wire.h>
#include "RTClib.h"
```

OK this is the top of the file, where we include the three libraries we'll use: the **SD** library to talk to the card, the **Wire** library that helps the Arduino with i2c and the **RTClib** for chatting with the real time clock

```
// A simple data logger for the Arduino analog pins
#define LOG_INTERVAL  1000 // mills between entries
#define ECHO_TO_SERIAL   1 // echo data to serial port
#define WAIT_TO_START    0 // Wait for serial input in setup()


// the digital pins that connect to the LEDs
#define redLEDpin 3
#define greenLEDpin 4

// The analog pins that connect to the sensors
#define photocellPin 0           // analog 0
#define tempPin 1                // analog 1
```

Next are all the "defines" - the constants and tweakables.

- `LOG_INTERVAL` is how many milliseconds between sensor readings. 1000 is 1 second which is not a bad starting point
- `ECHO_TO_SERIA`L determines whether to send the stuff thats being written to the card also out to the Serial monitor. This makes the logger a little more sluggish and you may want the serial monitor for other stuff. On the other hand, its hella useful. We'll set this to **1** to keep it on. Setting it to **0** will turn it off
- `WAIT_TO_START` means that you have to send a character to the Arduino's Serial port to kick start the logging. If you have this on you basically can't have it run away from the computer so we'll keep it off (set to **0**) for now. If you want to turn it on, set this to **1**

The other defines are easier to understand, as they are just pin defines

- `redLEDpin` is whatever you connected to the Red LED on the logger shield
- `greenLEDpin` is whatever you connected to the Green LED on the logger shield
- `photocellPin` is the analog input that the CdS cell is wired to
- `tempPin` is the analog input that the TMP36 is wired to

Objects and error()

```
RTC_DS1307 RTC; // define the Real Time Clock object


// for the data logging shield, we use digital pin 10 for the SD cs line
const int chipSelect = 10;

// the logging file
File logfile;

void error(char *str)
{
  Serial.print("error: ");
  Serial.println(str);

  // red LED indicates error
  digitalWrite(redLEDpin, HIGH);

  while(1);
}
```

Next up we've got all the objects for the RTC, and the SD card chip select pin. For all our shields we use **pin 10** for SD card chip select lines

Next is the `error()` function, which is just a shortcut for us, we use it when something Really Bad happened, like we couldn't write to the SD card or open it. It prints out the error to the Serial Monitor, turns on the red error LED, and then sits in a `while(1);` loop forever, also known as a **halt**

Setup

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

#if WAIT_TO_START
  Serial.println("Type any character to start");
  while (!Serial.available());
#endif //WAIT_TO_START
```

K now we are onto the code. We begin by initializing the Serial port at 9600 baud. If we set `WAIT_TO_START` to anything but **0**, the Arduino will wait until the user types something in. Otherwise it goes ahead to the next part

```
// initialize the SD card
Serial.print("Initializing SD card...");
// make sure that the default chip select pin is set to
// output, even if you don't use it:
pinMode(10, OUTPUT);

// see if the card is present and can be initialized:
if (!SD.begin(chipSelect)) {
  Serial.println("Card failed, or not present");
  // don't do anything more:
  return;
}
Serial.println("card initialized.");

// create a new file
char filename[] = "LOGGER00.CSV";
for (uint8_t i = 0; i < 100; i++) {
  filename[6] = i/10 + '0';
  filename[7] = i%10 + '0';
  if (! SD.exists(filename)) {
    // only open a new file if it doesn't exist
    logfile = SD.open(filename, FILE_WRITE);
    break;  // leave the loop!
  }
}

if (! logfile) {
  error("couldnt create file");
}

Serial.print("Logging to: ");
Serial.println(filename);
```

Now the code starts to talk to the SD card, it tries to initialize the card and find a FAT16/FAT32 partition.

Next it will try to make a logfile. We do a little tricky thing here, we basically want the files to be called something like **LOGGERnn.csv** where **nn** is a number. By starting out trying to create **LOGGER00.CSV** and incrementing every time when the file already exists, until we get to **LOGGER99.csv**, we basically make a new file every time the Arduino starts up

To create a file, we use some Unix style command flags which you can see in the `logfile.open()` procedure. **FILE_WRITE** means to create the file and write data to it.

Assuming we managed to create a file successfully, we print out the name to the Serial port.

```
  Wire.begin();
  if (!RTC.begin()) {
    logfile.println("RTC failed");
#if ECHO_TO_SERIAL
    Serial.println("RTC failed");
#endif  //ECHO_TO_SERIAL
  }


  logfile.println("millis,time,light,temp");
#if ECHO_TO_SERIAL
  Serial.println("millis,time,light,temp");
#if ECHO_TO_SERIAL// attempt to write out the header to the file
  if (logfile.writeError || !logfile.sync()) {
    error("write header");
  }

  pinMode(redLEDpin, OUTPUT);
  pinMode(greenLEDpin, OUTPUT);

   // If you want to set the aref to something other than 5v
  //analogReference(EXTERNAL);
}
```

OK we're wrapping up here. Now we kick off the RTC by initializing the Wire library and poking the RTC to see if its alive.

Then we print the header. The header is the first line of the file and helps your spreadsheet or math program identify whats coming up next. The data is in CSV (comma separated value) format so the header is too: "millis,time,light,temp" the first item **millis** is milliseconds since the Arduino started, **time** is the time and date from the RTC, **light** is the data from the CdS cell and **temp** is the temperature read.

You'll notice that right after each call to **logfile.print()** we have #if ECHO_TO_SERIAL and a matching **Serial.print()** call followed by a #if ECHO_TO_SERIAL this is that debugging output we mentioned earlier. The **logfile.print()** call is what writes data to our file on the SD card, it works pretty much the same as the **Serial** version. If you set **ECHO_TO_SERIAL** to be **0** up top, you won't see the written data printed to the Serial terminal.

Finally, we set the two LED pins to be outputs so we can use them to communicate with the user. There is a commented-out line where we set the analog reference voltage. This code assumes that you will be using the 'default' reference which is the VCC voltage for the chip - on a classic Arduino this is 5.0V. You can get better precision sometimes by lowering the reference. However we're going to keep this simple for now! Later on, you may want to experiment with it.

Main loop

Now we're onto the loop, the loop basically does the following over and over:

1. Wait until its time for the next reading (say once a second - depends on what we defined)
2. Ask for the current time and date froom the RTC
3. Log the time and date to the SD card
4. Read the photocell and temperature sensor
5. Log those readings to the SD card
6. Sync data to the card if its time

Timestamping

Lets look at the first section:

```
void loop(void)
{
  DateTime now;

  // delay for the amount of time we want between readings
  delay((LOG_INTERVAL -1) - (millis() % LOG_INTERVAL));

  digitalWrite(greenLEDpin, HIGH);

  // log milliseconds since starting
  uint32_t m = millis();
  logfile.print(m);           // milliseconds since start
  logfile.print(", ");
#if ECHO_TO_SERIAL
  Serial.print(m);         // milliseconds since start
  Serial.print(", ");
#endif

  // fetch the time
  now = RTC.now();
  // log time
  logfile.print(now.get()); // seconds since 2000
  logfile.print(", ");
  logfile.print(now.year(), DEC);
  logfile.print("/");
  logfile.print(now.month(), DEC);
  logfile.print("/");
  logfile.print(now.day(), DEC);
  logfile.print(" ");
  logfile.print(now.hour(), DEC);
  logfile.print(":");
  logfile.print(now.minute(), DEC);
  logfile.print(":");
  logfile.print(now.second(), DEC);
#if ECHO_TO_SERIAL
  Serial.print(now.get()); // seconds since 2000
  Serial.print(", ");
  Serial.print(now.year(), DEC);
  Serial.print("/");
  Serial.print(now.month(), DEC);
  Serial.print("/");
  Serial.print(now.day(), DEC);
  Serial.print(" ");
  Serial.print(now.hour(), DEC);
  Serial.print(":");
  Serial.print(now.minute(), DEC);
  Serial.print(":");
  Serial.print(now.second(), DEC);
#endif //ECHO_TO_SERIAL
```

The first important thing is the **delay()** call, this is what makes the Arduino wait around until its time to take another reading. If you recall we **#defined** the delay between readings to be 1000 millseconds (1 second). By having more delay between readings we can use less power and not fill the card as fast. Its basically a tradeoff how often you want to read data but for basic long term logging, taking data every second or so will result in plenty of data!

Then we turn the green LED on, this is useful to tell us that yes we're reading/writing data now.

Next we call **millis**() to get the 'time since arduino turned on' and log that to the card. It can be handy to have - especially if you end up not using the RTC.

Then the familiar **RTC.now()** call to get a snapshot of the time. Once we have that, we write a timestamp (seconods since 2000) as well as the date in **YY/MM/DD HH:MM:SS** time format which can easily be recognized by a spreadsheet. We have both because the nice thing about a timestamp is that its going to montonically increase and the nice thing about printed out date is its human readable

Log sensor data

Next is the sensor logging code

```
int photocellReading = analogRead(photocellPin);
  delay(10);
  int tempReading = analogRead(tempPin);

  // converting that reading to voltage, for 3.3v arduino use 3.3
  float voltage = (tempReading * 5.0) / 1024.0;
  float temperatureC = (voltage - 0.5) * 100.0 ;
  float temperatureF = (temperatureC * 9.0 / 5.0) + 32.0;

  logfile.print(", ");
  logfile.print(photocellReading);
  logfile.print(", ");
  logfile.println(temperatureF);
#if ECHO_TO_SERIAL
  Serial.print(", ");
  Serial.print(photocellReading);
  Serial.print(", ");
  Serial.println(temperatureF);
#endif //ECHO_TO_SERIAL

  digitalWrite(greenLEDpin, LOW);
}
```

This code is pretty straight forward, the processing code is snagged from our earlier tutorial. Then we just **print()** it to the card with a comma seperating the two

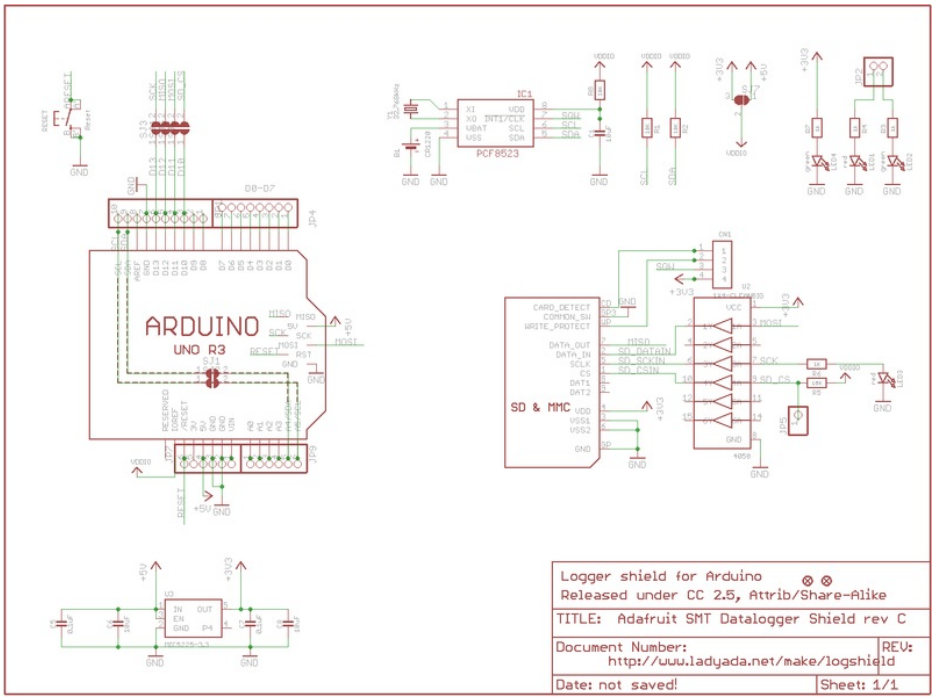
We finish up by turning the green LED off

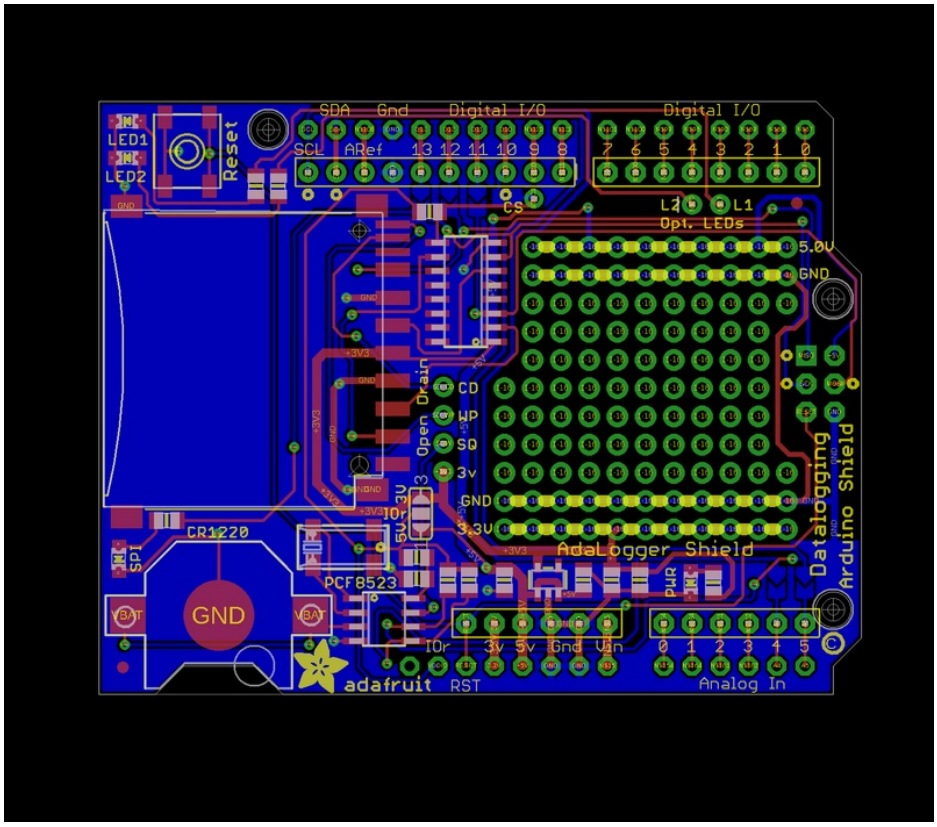
Downloads

Files

- EagleCAD PCB files on GitHub (https://adafru.it/rej)
- Fritzing object in Adafruit Fritzing library (https://adafru.it/aP3)

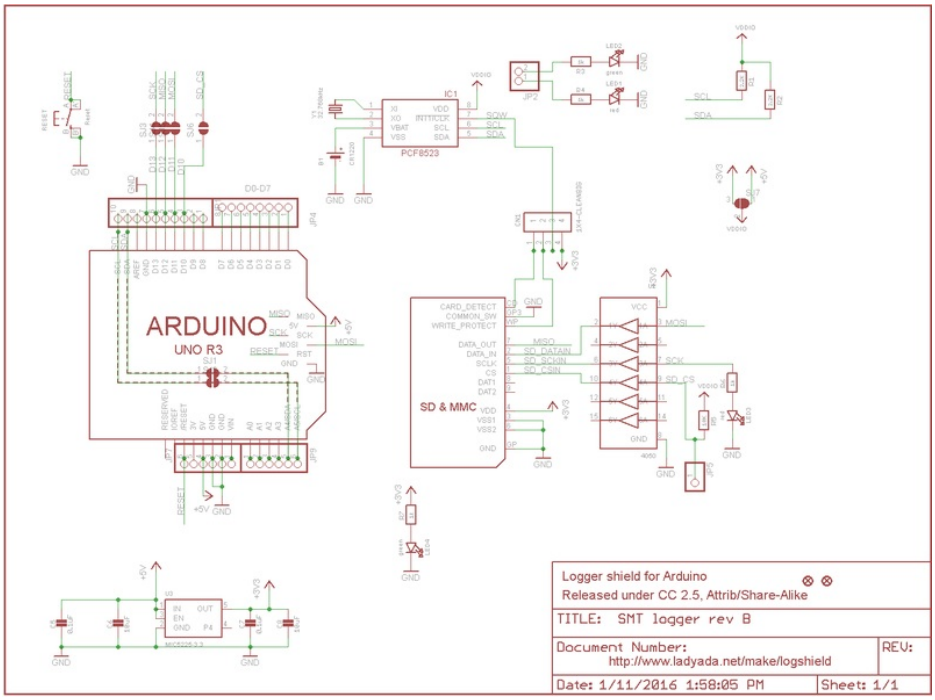
Revision C Schematics & Fabrication Print

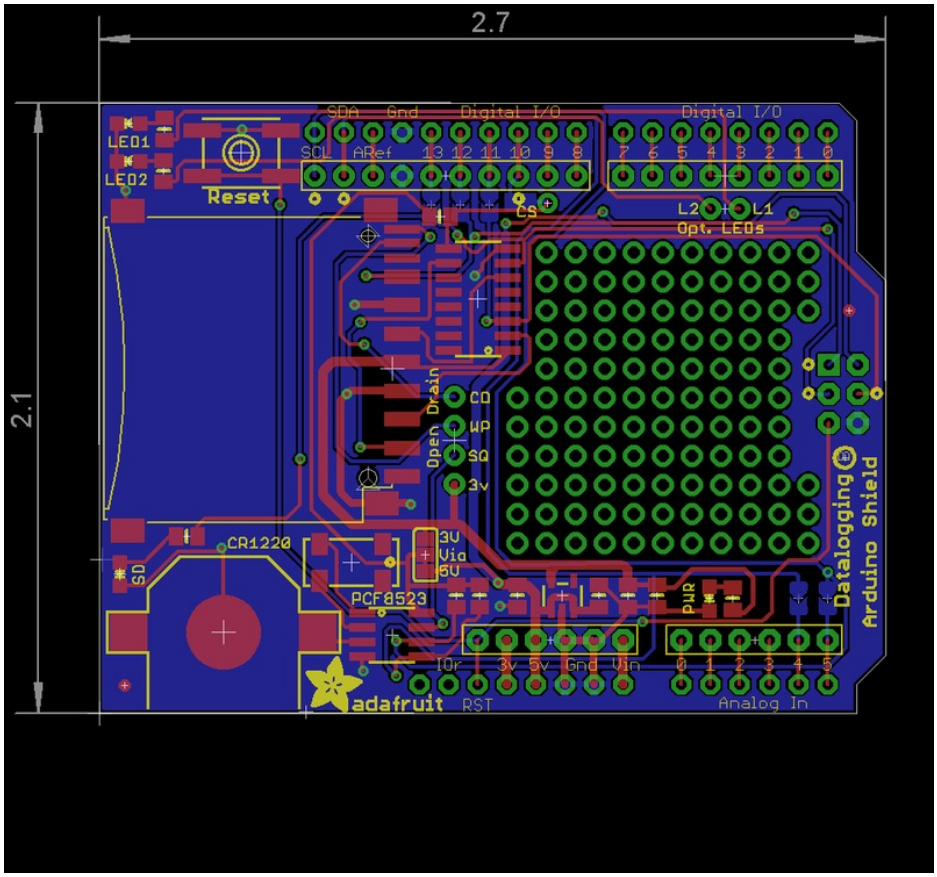




Revision B Schematics

click to enlarge





Original Version Schematics

Click to enlarge