Light and Temperature Logger 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!

Includes and Defines

```
#include "SdFat.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 **SdFat** 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()
#define SYNC_INTERVAL 1000 // mills between calls to sync()
uint32_t syncTime = 0;     // time of last sync()

// 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_SERIAL` 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**
- `SYNC_INTERVAL` - syncing is when we write the data permanently to the card including updating the filesize in the directory entry. You should sync pretty often (obviously you don't want to lose data) but there are times when you dont want to sync *too* often. For example, if you are recording audio data, the time required to sync would cause a hiccup in the sound. It might be better to wait until the chip is recorded and then sync. **In general**, though, you should sync as often as you write which is why this interval is also 1 second.

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

// The objects to talk to the SD card
Sd2Card card;
SdVolume volume;
SdFile root;
SdFile file;

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

```
  while(1);
}
```

Next up we've got all the objects for the RTC, and the SD card. The SD card actually has 4 objects because its complex enough! There's the **Sd2Card** object which is just for raw reading and writing.The **SdVolume** object looks for the FAT drive inside the card. **SdFiles** are both directories (**root** is the first directory) and files (**file** is well, our logging file)

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 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
  if (!card.init()) error("card.init");

  // initialize a FAT volume
  if (!volume.init(card)) error("volume.init");

  // open root directory
  if (!root.openRoot(volume)) error("openRoot");

  // create a new file
  char name[] = "LOGGER00.CSV";
  for (uint8_t i = 0; i < 100; i++) {
    name[6] = i/10 + '0';
    name[7] = i%10 + '0';
    if (file.open(root, name, O_CREAT | O_EXCL | O_WRITE)) break;
  }
```

```
  if (!file.isOpen()) error ("file.create");
  Serial.print("Logging to: ");
  Serial.println(name);

  // write header
  file.writeError = 0;
```

Now the code starts to talk to the SD card, it does so in a few steps, so that if there's a problem, the error printed out is somewhat useful. For example, if the card is completely missing, its a **Card Init** error. If there is no FAT filesystem, thats a **Volume Init** error. If for some reason it can't open the root directory that will have a different error (this is really unlikely but it is possible if the card is messed up)

Next it will try to make a file. 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 the call to create the file fails, 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 **file.open()** procedure. **O_CREAT** means to create this file. **O_EXCL** means to only create this file if it doesn't already exist (if you left this out, it would basically mean 'destroy any existing file, wipe it out'!), and **O_WRITE** means that we'll want to write to this file. If you're opening a file for reading, you'd leave this out which would keep you from accidentally modifying it.

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

`file.writeError` is a little ticker that lets us keep track of any problems we've had with the file. Since its a new file, we set it to 0 and hope it stays that way!

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


  file.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 (file.writeError || !file.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 **file.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 **file.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;

  // clear print error
  file.writeError = 0;

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

  digitalWrite(redLEDpin, HIGH);

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

  // fetch the time
  now = RTC.now();
  // log time
  file.print(now.get()); // seconds since 2000
  file.print(", ");
  file.print(now.year(), DEC);
  file.print("/");
  file.print(now.month(), DEC);
  file.print("/");
  file.print(now.day(), DEC);
  file.print(" ");
  file.print(now.hour(), DEC);
  file.print(":");
  file.print(now.minute(), DEC);
  file.print(":");
  file.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 red LED on, this is useful to tell us that yes we're reading 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;
  float temperatureC = (voltage - 0.5) * 100 ;
  float temperatureF = (temperatureC * 9 / 5) + 32;

  file.print(", ");
  file.print(photocellReading);
  file.print(", ");
  file.println(temperatureF);
#if ECHO_TO_SERIAL
  Serial.print(", ");
```

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

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

Sync writes

Now time to wrap up the sketch

```
  if (file.writeError) error("write data");

  digitalWrite(redLEDpin, LOW);

  //don't sync too often - requires 2048 bytes of I/O to SD card
  if ((millis() - syncTime) <  SYNC_INTERVAL) return;
  syncTime = millis();

  // blink LED to show we are syncing data to the card & updating
FAT!
  digitalWrite(greenLEDpin, HIGH);
  if (!file.sync()) error("sync");
  digitalWrite(greenLEDpin, LOW);
}
```

The first line checks for errors - if we had problems writing to the card (perhaps its out of space?) we'll stop logging

Then we check if its time to sync. As mentioned before, we only permanently write the data when syncing, so we want to sync often. But syncing also takes a lot of time so if we have to log data quickly, it might be beneficial to sync less often. We use the green LED to indicate that we're syncing, useful for debugging

June 3, 2010 20:25