SMARTbuoy

Water Quality Probe

Montgomery County Community College

May 5, 2018

Jeremy Reimert

Montgomery County Community College

Contents

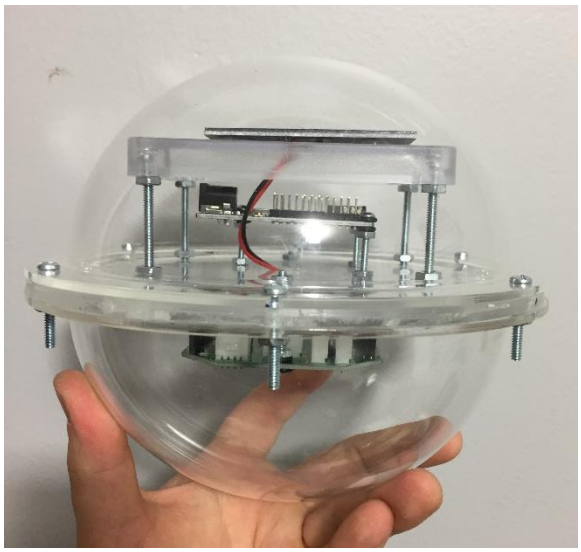
Introduction

The SMARTbouy was designed to be a fully self-contained water quality probe, able to be deployed into a body of water and monitored remotely via a GSM cellular network. The SMARTbuoy is an economical solution to water monitoring, meant for both amateur and professional researchers.

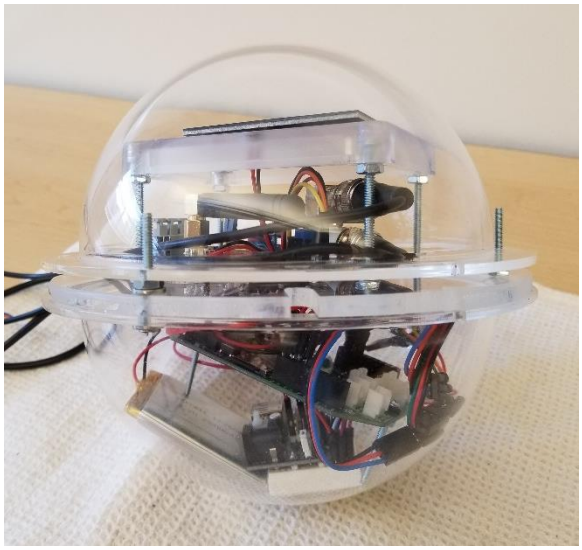
Our current build includes a 3.7volt lithium ion battery, charged using a solar panel and provides data for electrical conductivity, pH, temperature, turbidity, and total dissolved solids. Battery level and GPS data are also provided. Each reading is given a date and time stamp and recorded to an onboard SD card in CSV format. Additionally, each reading is printed on a serial monitor when attached to a computer via USB. Access to the cellular network is provided by a SIM900 series modem, and readings are "dweeted" using the dweet.io API. We have created a freeboard.io dashboard as a GUI which pulls data from dweet.io.

Future additions to the build will include more advanced sensors and connectivity between a network of SMARTbuoys. We also researched smartphone dashboard apps for mobile access and found Blynk to be a suitable candidate.

Project Photos



Exterior – before final assembly

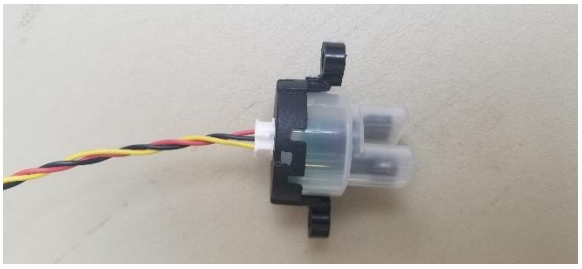


Exterior – fully assembled

Project Photos

pH Sensor

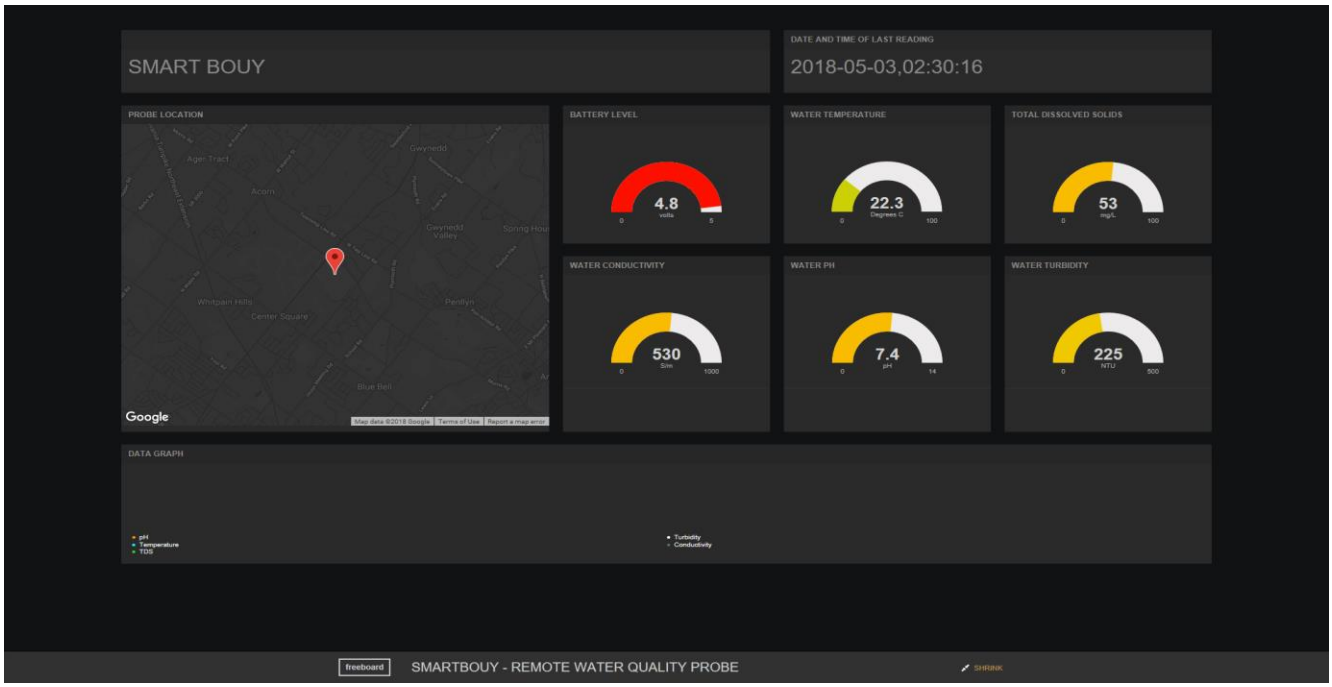
Electrical Conductivity Sensor



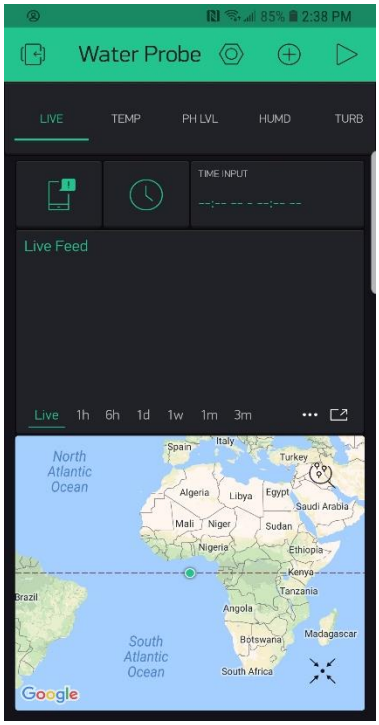
Turbidity Sensor

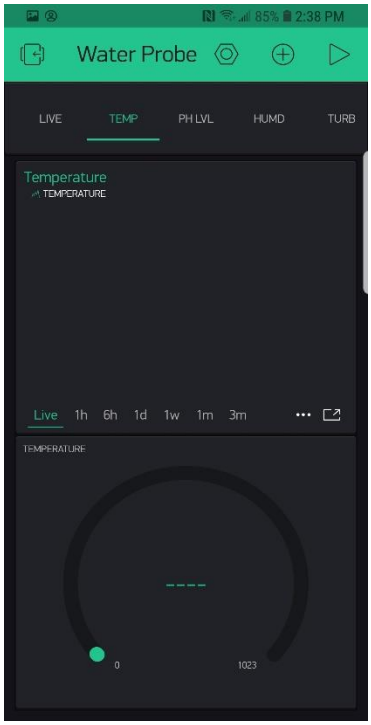
Temperature Sensor

GUI Elements



Freeboard.io Dashboard





Blynk app (incomplete)

External API

Dweet.io

Dweet.io allows users to share data from mobile, tablets, and pcs, and them to other devices and accounts across social media platforms. Dweet.io provides an API to access the different functionality of the Dweet.io service. Users can make REST calls to read and create dweets, lock and unlock things, and perform other calls. The API returns JSON and JSONP. Pre-built libraries are available for use in JavaScript and Node.js.

SPECS

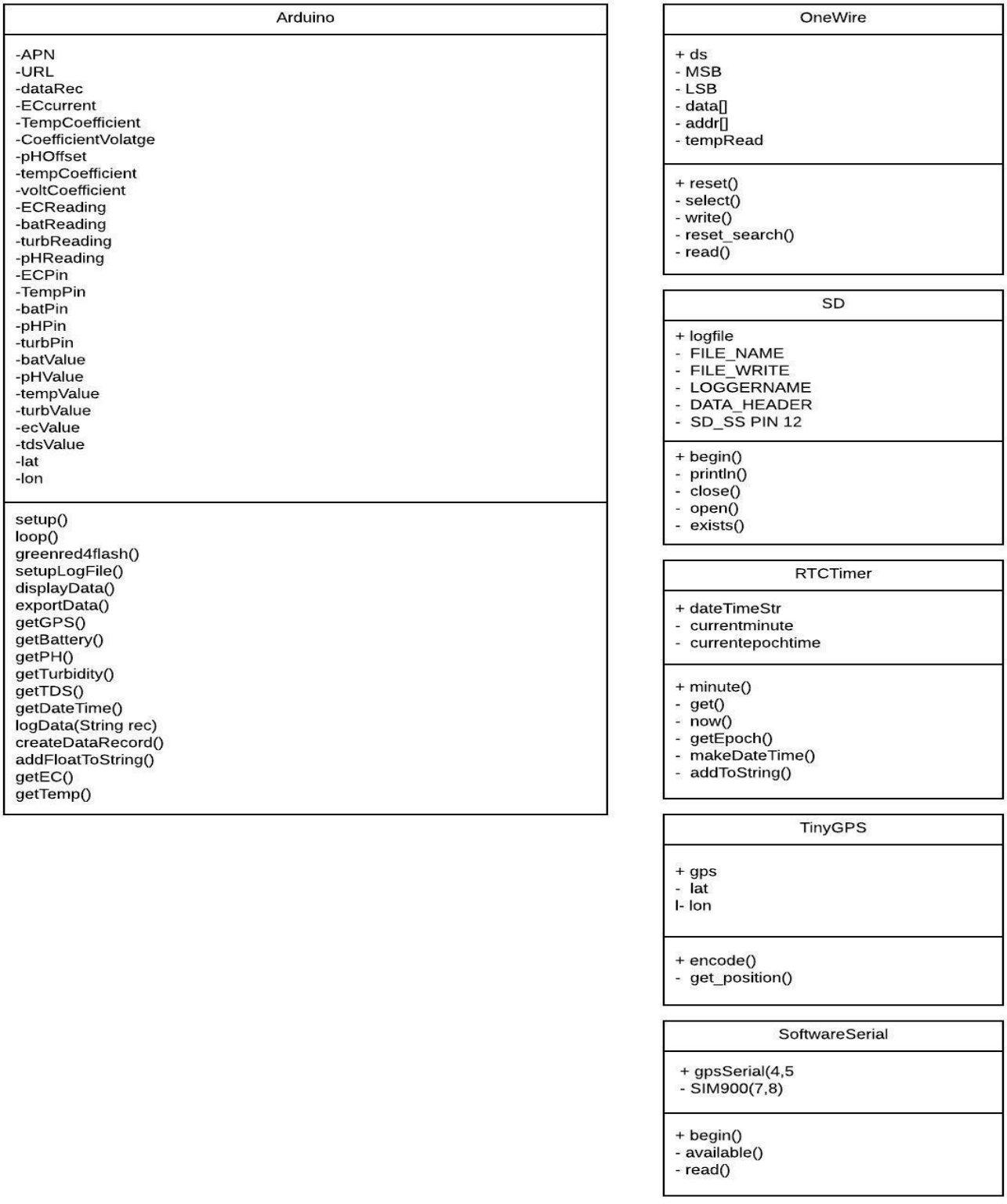
API Portal / Home Page https://dweet.io
Primary Category Social
Secondary Categories Notifications, Data
Support Email Address hello@dweet.io
Developer Support URL hello@dweet.io
Is the API Design/Description Non-Proprietary ? No
Scope Single purpose API
Device Specific No
Docs Home Page URL https://dweet.io
Architectural Style REST
Supported Request Formats JSONP, URI Query String/CRUD
Supported Response Formats JSON, JSONP
Is This an Unofficial API? No
Is This a Hypermedia API? Yes
Restricted Access (Requires Provider Approval) No

Links

Link to Github

https://github.com/jeremyreimert/SmartBuoy

UML Diagram


Arduino
-APN
-URL
-dataRec
-ECcurrent
-TempCoefficient
-CoefficientVolatge
-pHOffset
-tempCoefficient
-voltCoefficient
-ECReading
-batReading
-turbReading
-pHReading
-ECPin
-TempPin
-batPin
-pHPin
-turbPin
-batValue
-pHValue
-tempValue
-turbValue
-ecValue
-tdsValue
-lat
-lon
setup()
loop()
greenred4flash()
setupLogFile()
displayData()
exportData()
getGPS()
getBattery()
getPH()
getTurbidity()
getTDS()
getDateTime()
logData(String rec)
createDataRecord()
addFloatToString()
getEC()
getTemp()
OneWire
+ ds
- MSB
- LSB
- data[]
- addr[]
- tempRead
+ reset()
- select()
- write()
- reset_search()
- read()
SD
+ logfile
- FILE_NAME
- FILE_WRITE
- LOGGERNAME
- DATA_HEADER
- SD_SS PIN 12
+ begin()
- println()
- close()
- open()
- exists()
RTCTimer
+ dateTimeStr
- currentminute
- currentepochtime
+ minute()
- get()
- now()
- getEpoch()
- makeDateTime()
- addToString()
TinyGPS
+ gps
- lat
I- lon
+ encode()
- get_position()
SoftwareSerial
+ gpsSerial(4,5
- SIM900(7,8)
+ begin()
- available()
- read()

Code

```
/* SMART BOUY
 * A completely self-contained, fully deployable remote water quality sensor
 *
 * Features:  pH sensor
 *          Turbidity Sensor
 *          Temperature Sensor
 *          Electrical Conductivity Sensor
 *          Total Dissolved Solids (derived from conductivity)
 *          Date and Time Stamp for each reading
 *          GPS tracking
 *          Battery level indicator
 *          Solar Panel - charges onboard 3.7 volt lithium ion battery
 *          Data upload through GSM cellular network (2G)
 *          On board data logging to a micro SD card
 *
 * Process:
 * 1. Each sensor reading is taken and stored
 * 2. A string is bulit from the sensor data and stored on the SD card in CSV format
 * 3. Sensor readings are sent to a serial monitor
 * 4. A URL string is built from the data
 * 5. An http GET request including the URL string is sent to dweet.io API after each reading
 * 6. A freeboard dashboard has been created to pull data from dweet.io and display data
graphically https://freeboard.io/board/fgJLRa
*/

#include <OneWire.h>            // library for temp sensor, allows multiple sensors on a single pin
#include <SPI.h>               // Serial Peripheral Interface - for communicating with peripheral
devices
#include <SD.h>                // for reading/writing SD card
#include <RTCTimer.h>           // library for real time clock
#include <Sodaq_DS3231.h>        // library for on board clock
#include <SoftwareSerial.h>      // library allowing digital pins to act as communcation pins
#include <TinyGPS.h>            // GPS library


String  APN = "wholesale";       // apn for http network access
String  URL;                  // website to be accesses
String  dateTimeStr;            // hold date and time
String  dataRec = "";           // string for data added to SD card
int     currentminute;          // current time
```

```
long   currentepochtime = 0;     // current time

float  ECcurrent;              // converted EC reading
float  TempCoefficient;         // used for data normalization
float  CoefficientVolatge;      // used for data normalization
float  pHOffset = -.48;         // offset for normalizing readings
float  tempCoefficient;         // used for EC conversion
float  voltCoefficient;         // used for EC conversion

float  ECReading = 0;           // variable to store the value coming from the analogRead function
int    batReading = 0;          // variable to store the value coming from the analogRead function
int    turbReading = 0;         // variable to store the value coming from the analogRead function
int    pHReading = 0;           // variable to store the value coming from the analogRead function

byte   ECPin = A1;              // analog pin 1 - input from conductivity meter
byte   TempPin = 6;             // digital pin 6 - input from thermometer
int    batPin = A6;             // analog pin 6 - input from battery
int    pHPin = A5;              // analog pin 5 - input from pH meter
int    turbPin = A3;            // analog pin 3 - input from turbidity meter

float  batValue;                // the voltage
float  pHValue;                 // the pH
float  tempValue;               // the temperature
float  turbValue;               // the turbitity
float  ecValue;                 // the conductivity
float  tdsValue = 0;            // the total dissolved solids
long   lat;                     // the latitude
long   lon;                     // the longitude

#define SD_SS_PIN 12            // pin used to write SD card
#define FILE_NAME "datafile.txt"   // name of data file
#define LOGGERNAME "DataLogger"    // title of data file table
#define DATA_HEADER "DateTime_EST,Battery_V,Temperature_C,pH
Level,Turbidity_NTU,Conductivity_us/cm,TDS_mg/L"  // format for data file table header

OneWire ds(TempPin);            // create a OneWire object on digital pin 4
SoftwareSerial gpsSerial(4, 5);   // create a SoftwarSerial object for gps sensor connection on pins
4(rx) & 5(tx)
SoftwareSerial SIM900(7,8);       // create a SoftwarSerial object for modem connection on pins
7(rx) & 8(tx)
TinyGPS gps;                    // create gps object

void setup()
{
      Serial.begin(9600);    // initialize serial monitor
```

```
        gpsSerial.begin(9600); // initialize gps sensor
        SIM900.begin(19200);   // initialize SIM900 modem
        rtc.begin();         // initialize real time clock
        delay(100);

        pinMode(8, OUTPUT);
        pinMode(9, OUTPUT);

        greenred4flash();    //blink the LEDs to show the board is on
        setupLogFile();      //checks for file - creates file and formats header

        // header for serial output
        Serial.println("Water probe Project: CIS 111B");
        Serial.println("");
        Serial.println("Initiating sensor readings and logging data to SDcard....");
        Serial.println("");

}

void loop()
{
        getDateTime();             // gets the current time and date
        getTemp();               // gets data from temperature sensor
        getEC();                // gets data from conductivity sensor
        getBattery();             // gets data from battery sensor
        getPH();                // gets data from pH sensor
        getTurbidity();            // gets data from turbidity sensor
        getTDS();               // converts data from conductivity sensor
        getGPS();               // gets GPS coordinates
        createDataRecord();         // assign the data string to dataRec
        createURL();             // builds the URL string from sensor data
        logData(dataRec);          // Save the data record to the log file
        displayData();           // Send the data to the serial connection
        exportData();            // Send the data to the  serial1 connection

        delay(10000);            //10 second delay
}

// prints data to serial monitor
void displayData()
{

Serial.println("=================================================================
================================================================================
=");
```

```
        Serial.println(" Date     Time     Battery     Temperature      pH Level     Turbidity
Conductivity     TDS          Latitude      Longitude      ");
        Serial.println(" (y/m/d)   (h/m/s)   (volts)     (degree C)                (NTU)        (us/cm)
(mg/L)                                ");
        Serial.print(dateTimeStr);
        Serial.print("\t");
        Serial.print(batValue);
        Serial.print("\t\t");
        Serial.print(tempValue);
        Serial.print("\t\t");
        Serial.print(pHValue);
        Serial.print("\t\t");
        Serial.print(turbValue);
        Serial.print("\t\t");
        Serial.print(ecValue);
        Serial.print("\t\t");
        Serial.print(tdsValue);
        Serial.print("\t\t");
        Serial.print(lat);
        Serial.print("\t");
        Serial.println(lon);

Serial.println("=============================================================================
=============================================================================
=");
}

void exportData()
{
        SIM900.println("AT+CPAS");
        delay(100);
        SIM900.read();

        SIM900.println("AT+CSQ");
        delay(100);
        SIM900.read();

        SIM900.println("AT+CGATT?");
        delay(100);
        SIM900.read();

        SIM900.println("AT+SAPBR=3,1,\"CONTYPE\",\"GPRS\"");  //setting the SAPBR, the
connection type is using gprs
        delay(1000);
        SIM900.read();
```

```
        SIM900.println("AT+SAPBR=3,1,\"APN\"," + APN));       //setting the APN, fill in your local apn
server
        delay(4000);
        SIM900.read();

        SIM900.println("AT+SAPBR=1,1");                  //setting the SAPBR
        delay(2000);
        SIM900.read();

        SIM900.println("AT+HTTPINIT");                  //init the HTTP request
        delay(2000);
        SIM900.read();

        SIM900.println("AT+HTTPPARA=\"URL\"," + URL);       // setting the website you want to
access
        delay(1000);
        SIM900.read();

        SIM900.println("AT+HTTPACTION=0");               //submit the request
        delay(10000);                              // longer delay allows time for data return from website
        SIM900.read();

        SIM900.println("AT+HTTPREAD");                  // read the data from the website you access
        delay(300);
        SIM900.read();

        SIM900.println("");
        delay(100);

}

void getGPS()
{
        while(gpsSerial.available())              // check for gps data
          {
           if(gps.encode(gpsSerial.read()))     // encode gps data
             {
                gps.get_position(&lat,&lon);    // get latitude and longitude
             }
          }
}
```

```
//get the date and time of the reading
void getDateTime()
{
        dateTimeStr = "";
        //Create a DateTime object from the current time
        DateTime dt(rtc.makeDateTime(rtc.now().getEpoch()));
        currentepochtime = (dt.get());
        currentminute = (dt.minute());
        dt.addToString(dateTimeStr); //Convert it to a String

}


// flashes LEDs to confirm power
void greenred4flash()
{
        for (int i=1; i <= 4; i++)
        {
           digitalWrite(8, HIGH);
           digitalWrite(9, LOW);
           delay(50);
           digitalWrite(8, LOW);
           digitalWrite(9, HIGH);
           delay(50);
        }
        digitalWrite(9, LOW);
}


// confirm file exists then adds title and header
void setupLogFile()
{
        //Initialize the SD card
         if (!SD.begin(SD_SS_PIN))
         {
           Serial.println("Error: SD card failed to initialize or is missing.");
         }

        //Check if the file already exists
        bool oldFile = SD.exists(FILE_NAME);

        //Open the file in write mode
        File logFile = SD.open(FILE_NAME, FILE_WRITE);

        //Add header information if the file did not already exist
```

```
        if (!oldFile)
        {
           logFile.println(LOGGERNAME);
           logFile.println(DATA_HEADER);
        }

        //Close the file to save it
        logFile.close();
}


// adds data to SD card
void logData(String rec)
{
        //Re-open the file
        File logFile = SD.open(FILE_NAME, FILE_WRITE);

        //Write the CSV data
        logFile.println(rec);

        //Close the file to save it
        logFile.close();
}


//creates the URL string for data upload to dweet.io API
void createURL()
{
        //Create a url string
        String urlString = "http://www.dweet.io/dweet/for/WaterProbe111B?"; // beginning of url
        urlString += "BATT=";
        addFloatToString(urlString, batValue, 3, 1);       //add battery data
        urlString += "&PH=";
        addFloatToString(urlString, pHValue, 4, 2);        //add temperature data
        urlString += "&TURB=";
        addFloatToString(urlString, turbValue, 3, 1);      //add pH data
        urlString += "&ECOND=";
        addFloatToString(urlString, ecValue, 3, 1);        //add turbidity data
        urlString += "&TDS=";
        addFloatToString(urlString, tdsValue, 3, 1);       //add conductivity data
        urlString += "&TEMP=";
        addFloatToString(urlString, tempValue, 3, 1);      //add TDS data
        urlString += "&LON=";
        addFloatToString(urlString, lat, 1, 6);            //add lat data
        urlString += "&LAT=";
```

```
        addFloatToString(urlString, lon, 1, 6);          //add lon data
        urlString += "&DT=";
        urlString += dateTimeStr;                        //add date and time
        URL = urlString;
        Serial.println(URL);
}


//creates the data string to hold all sensor readings
void createDataRecord()
{
        //Create a String type data record in csv format
        String data = dateTimeStr;               //add date and time data
        data += ",";
        addFloatToString(data, batValue, 3, 1);     //add battery data
        data += ",";
        addFloatToString(data, tempValue, 4, 2);    //add temperature data
        data += ",";
        addFloatToString(data, pHValue, 3, 1);      //add pH data
        data += ",";
        addFloatToString(data, turbValue, 3, 1);    //add turbidity data
        data += ",";
        addFloatToString(data, ecValue, 3, 1);      //add conductivity data
        data += ",";
        addFloatToString(data, tdsValue, 3, 1);     //add TDS data
        data += ",";
        addFloatToString(data, lat, 1, 6);          //add lat data
        data += ",";
        addFloatToString(data, lon, 1, 6);          //add lon data
        dataRec = data;
}


//converts sensor data to a string
static void addFloatToString(String & str, float val, char width, unsigned char precision)
{
        char buffer[10];
        dtostrf(val, width, precision, buffer);
        str += buffer;
}


//gets battery reading and converts to usable data
void getBattery()
{
```

```
        batReading = analogRead(batPin);            // reads input from meter
        batValue = batReading * (15.75/1024);       // convert the analog input
}

//gets pH reading and converts to usable data
void getPH()
{

        pHReading = analogRead(pHPin);              // reads input from meter
        pHValue = (pHReading * (17.5/6144)) + pHOffset; // convert the analog input
}

//gets turbidity reading and converts to usable data
void getTurbidity()
{
        turbReading = analogRead(turbPin);          // reads input from meter
        turbValue = turbReading * (5/1024);         // convert the analog input
        if(turbValue < 0)
          turbValue = 0;
}

//converts the conductivity reading to TDS
void getTDS()
{
        tdsValue = ecValue * .67;
}

// gets the conductivity reading
void getEC()
{
        if(ECReading<=448)
            ECcurrent=(6.84*ECReading-64.32);   //1ms/cm<EC<=3ms/cm
        else

        if(ECReading<=1457)
            ECcurrent=(6.98*ECReading-127);     //3ms/cm<EC<=10ms/cm
        else
            ECcurrent=(5.3*ECReading+2278);     //10ms/cm<EC<20ms/cm

        ecValue = ECcurrent/1000;

        if(ecValue < 0)
            ecValue = 0;
}
```

```
//gets temperature from one tempPin in DEG Celsius
void getTemp()
{
	byte data[12];
	byte addr[8];

	if ( !ds.search(addr))
	{
	   //no sensors, reset search
	   ds.reset_search();
	   tempValue = -1000;
	}

	if ( OneWire::crc8( addr, 7) != addr[7])
	{
	    tempValue = -1000;
	}

	if ( addr[0] != 0x10 && addr[0] != 0x28)
	{
	   tempValue = -1000;
	}

	ds.reset();
	ds.select(addr);
	ds.write(0x44,1); // start conversion, with parasite power on at the end

	byte present = ds.reset();
	ds.select(addr);
	ds.write(0xBE); // Read Scratchpad

	for (int i = 0; i < 9; i++) // we need 9 bytes
	{
	   data[i] = ds.read();
	}

	ds.reset_search();

	byte MSB = data[1];
	byte LSB = data[0];

	float tempRead = ((MSB << 8) | LSB); //using two's compliment
	tempValue = tempRead / 16;
}
```

Documentation

SD Library

The SD library allows for reading from and writing to SD cards, e.g. on the Arduino Ethernet Shield. It is built on sdfatlib by William Greiman. The library supports FAT16 and FAT32 file systems on standard SD cards and SDHC cards. It uses short 8.3 names for files. The file names passed to the SD library functions can include paths separated by forward-slashes, /, e.g. "directory/filename.txt". Because the working directory is always the root of the SD card, a name refers to the same file whether or not it includes a leading slash (e.g. "/file.txt" is equivalent to "file.txt"). As of version 1.0, the library supports opening multiple files.

The communication between the microcontroller and the SD card uses SPI, which takes place on digital pins 11, 12, and 13 (on most Arduino boards) or 50, 51, and 52 (Arduino Mega). Additionally, another pin must be used to select the SD card. This can be the hardware SS pin - pin 10 (on most Arduino boards) or pin 53 (on the Mega) - or another pin specified in the call to SD.begin(). *Note that even if you don't use the hardware SS pin, it must be left as an output or the SD library won't work.*

Notes on using the Library and various shields

Examples

Card Info: Get info about your SD card.
Datalogger: Log data from three analog sensors to an SD card.
Dump File: Read a file from the SD card.
Files: Create and destroy an SD card file.
List Files: Print out the files in a directory on a SD card.
Read Write: Read and write data to and from an SD card.

SD class

The SD class provides functions for accessing the SD card and manipulating its files and directories.

- begin()
- exists()
- mkdir()
- open()
- remove()
- rmdir()

File class

The File class allows for reading from and writing to individual files on the SD card.

- name()
- available()
- close()
- flush()
- peek()
- position()
- print()
- println()
- seek()
- size()
- read()
- write()
- isDirectory()
- openNextFile()
- rewindDirectory()

begin()

Description

Initializes the SD library and card. This begins use of the SPI bus (digital pins 11, 12, and 13 on most Arduino boards; 50, 51, and 52 on the Mega) and the chip select pin, which defaults to the hardware SS pin (pin 10 on most Arduino boards, 53 on the Mega). Note that even if you use a different chip select pin, *the hardware SS pin must be kept as an output* or the SD library functions will not work.

Syntax

SD.begin()
SD.begin(cspin)

Parameters

cspin (*optional*): the pin connected to the chip select line of the SD card; defaults to the hardware SS line of the SPI bus

exists()

Description

Tests whether a file or directory exists on the SD card.

Syntax

SD.exists(filename)

Parameters

filename: the name of the file to test for existence, which can include directories (delimited by forward-slashes, /)

Returns

true if the file or directory exists, false if not

close()

Close the file, and ensure that any data written to it is physically saved to the SD card.

Syntax

***file*.close()**

Parameters

***file*: an instance of the File class (returned by SD.open())**

Returns

none

println()

Description

Print data, followed by a carriage return and newline, to the File, which must have been opened for writing. Prints numbers as a sequence of digits, each an ASCII character (e.g. the number 123 is sent as the three characters '1', '2', '3').

Syntax

***file*.println()**
***file*.println(data)**
***file*.print(data, BASE)**

Parameters

***file*: an instance of the File class (returned by SD.open())**

data (optional): the data to print (char, byte, int, long, or string)

BASE (optional): the base in which to print numbers: BIN for binary (base 2), DEC for decimal (base 10), OCT for octal (base 8), HEX for hexadecimal (base 16).

Returns

byte
println() will return the number of bytes written, though reading that number is optional

open()

Description

Opens a file on the SD card. If the file is opened for writing, it will be created if it doesn't already exist (but the directory containing it must already exist).

Syntax

SD.open(filepath)
SD.open(filepath, mode)

Parameters

filename: the name the file to open, which can include directories (delimited by forward slashes, /) - *char **

mode (*optional*): the mode in which to open the file, defaults to FILE_READ - *byte*. one of:

FILE_READ: open the file for reading, starting at the beginning of the file.

FILE_WRITE: open the file for reading and writing, starting at the end of the file.

Returns

a File object referring to the opened file; if the file couldn't be opened, this

OneWire

Latest version

The latest version of the library is on Paul Stoffregen's site. The rest of this page is a work in progress.

OneWire is currently maintained by Paul Stoffregen. If you find a bug or have an improvement (to the library), email paul at pjrc dot com. Please be sure you are using the latest version of OneWire.

Bus is a subclass of the OneWire library. Bus class scans the 1 wire Bus connected to an arduino UNO analog pin and stores the ROMs in an array. Several methods are available in the Bus class to acquire datas from by different 1wire sensors (DS18B20, DS2438).

The 1-Wire Protocol

Dallas Semiconductor (now Maxim) produces a family of devices that are controlled through a proprietary **1-wire** protocol. There are no fees for programmers using the Dallas 1-Wire (trademark) drivers.

On a 1-Wire network, which Dallas has dubbed a "MicroLan" (trademark), a single "master" device communicates with one or more 1-Wire "slave" devices over a single data line, which can also be used to provide power to the slave devices. (Devices drawing power from the 1-wire bus are said to be operating in *parasitic power* mode.) http://sheepdogguides.com/arduino/asw1onew1.htm Tom Boyd's guide to 1-Wire may tell you more than you want to know... but it may also answer questions and inspire interest.

The 1-wire temperature sensors have become particularly popular, because they're inexpensive and easy to use, providing calibrated digital temperature readings directly. They are more tolerant of long wires between sensor and Arduino. The sample code below demonstrates how to interface with a 1-wire device using Jim Studt's *OneWire* Arduino library, with the DS18S20 digital thermometer as an example. Many 1-Wire chips can operate in both parasitic and normal power modes.

1Wire Interfaces

Dedicated Bus Masters

Dallas/Maxim and a number of other companies manufacture dedicated bus masters for read/write and management of 1Wire networks. Most of these are listed here:

http://owfs.org/index.php?page=bus-masters

These devices are specifically designed and optimized to read and write efficiently to 1Wire devices and networks. Similar to UART/USART masters, they handle clocked operations natively with the use of a buffer, offloading the processing load from the host processor (e.g. sensor gateway or microcontroller) and increase accuracy. External pull-up resistors are also often not required.

Many of the chips provide error-handling that specifically deals with loss of signal integrity, level variation, reflections, and other bus issues that may cause problems, particularly on large networks. Many of the devices have additional features, and are offered on a large variety of interfaces. They range in price from $1 to $30.

Another key advantage is support of , a read/write file system with vast device support for 1Wire masters that exposes many native functions for a wide variety of 1Wire device types.

UART/USART Masters

Most UART/USARTs are perfectly capable of sustained speeds well in excess of the 15.4kbps required of the 1Wire bus in standard mode. More important, the clock and buffering is handled separately, again offloading it from the main process of the microcontroller or main processor. This implementation is discussed here: http://www.maximintegrated.com/en/app-notes/index.mvp/id/214

Bitbanging approaches

Where native buffering/clock management is not available, 1Wire may be implemented on a general purpose IO (GPIO) pin, where manual toggle of the pin state is used to emulate a UART/USART with reconstruction of the signal from the received data. These are typically much less processor-efficient, and directly impact and are directly impacted by other processes on the processor shared with other system processes.

On Arduino and other compatible chips, this may be done with the OneWire library (linked above, examples below) on any available digital pin.

On single-board computers such as the Raspberry Pi, 1Wire network read is often possible using kernel drivers that offer native support. The w1-gpio, w1-gpio-therm, and w1-gpio-custom kernel mods are included in the most recent distributions of Raspbian and are quite popular, as they allow interfacing with a subset of 1Wire device with no additional hardware. Currently, however, they have limited device support, and have bus size limitations in software.

Powering OneWire devices

The chip can be powered two ways. One (the "parasitic" option) means that only two wires need go to the chip. The other may, in some cases, give more reliable operation (parasitic often works well), as an extra wire carrying the power for the chip is involved. For getting started, especially if your chip is within 20 feet of your Arduino, the parasitic option is probably fine. The code below works for either option, anyway.

Parasite power mode
When operating in parasite power mode, only two wires are required: one data wire, and ground. In this mode, the power line must be connected to ground, per the datasheet. At the master, **a 4.7k pull-up resistor must be connected to the 1-wire bus.** When the line is in a "high" state, the device pulls current to charge an internal capacitor.

This current is usually very small, but may go as high as 1.5 mA when doing a temperature conversion or writing EEPROM. When a slave device is performing one these operations, the bus master must keep the bus pulled high to provide power until the operation completes; a delay of 750ms is required for a DS18S20 temperature conversion. The master can't do anything during this time, like issuing commands to other devices, or polling for the slave's operation to be completed. To support this, the OneWire library makes it possible to have the bus held high after the data is written.

Normal (external supply) mode
With an external supply, three wires are required: the bus wire, ground, and power. **The 4.7k pull-up resistor is still required** on the bus wire. As the bus is free for data transfer, the microcontroller can continually poll the state of a device doing a conversion. This way, a conversion request can finish as soon as the device reports being done, as opposed to having to wait for conversion time (dependent on device function and resolution) in "parasite" power mode.

Note on resistors:
For larger networks, you can try smaller resistors.
The ATmega328/168 datasheet indicates starting at 1k6 and a number of users have found smaller to work better on larger networks.

Addressing a OneWire device

Each 1-Wire device contains a unique 64-bit 'ROM' address, consisting of an 8-bit family code, a 48-bit serial number, and an 8-bit CRC. The CRC is used to verify the integrity of the data. For example, the sample code, below, checks if the device being addressed is a DS18S20 temperature sensor by checking for its family code, 0x10. To use the sample code with the newer DS18B20 sensor, you'd check for a family code of 0x28, instead, and for the DS1822 you'd check for 0x22.

Single-device commands
Before sending a command to a single slave device, the master must first select that device using its unique ROM. Subsequent commands will be responded to by the selected device, if found.

Multiple-device commands
Alternatively, you can address a command to all slave devices by issuing a 'Skip ROM' command (0xCC), instead. It is important to consider the effects of issuing a command to multiple devices. Sometimes, this may be intended and beneficial. For example, issuing a Skip ROM followed by a convert T (0x44) would instruct all networked devices that have a Convert T command to perform a temperature conversion. This can be a time-saving and efficient way of performing the operations. On the other hand, issuing a Read Scratchpad (0xBE) command would cause all devices to report

Scratchpad data simultaneously. Power consumption of all devices (for example, during a temperature conversion) is also important when using a Skip ROM command sequence.

Please see the DS18S20 or DS18B20 datasheets for more detailed information.

Reading a OneWire device

Reading a 1Wire device requires multiple steps. The details are device-dependent, in that devices are capable of reporting different measurables. The popular DS18B20, for example, reads and reports temperature, while a DS2438 reads voltage, current, and temperature.

Two Main Read Process Steps:

Conversion

A command is issued to the device to perform an internal conversion operation. With a DS18B20, this is the Convert T (0x44) byte command. In the OneWire library, this is issued as ds.write(0x44), where ds is an instance of the OneWire class. After this command is issued, the device reads the internal ADC,and when this process is complete, it copies the data into the Scratchpad registers. This length of this conversion process varies depending on the resolution and is listed in the device datasheet. a DS18B20 takes from 94 (9-bit resolution) to 750ms (12-bit resolution) to convert temperature (c.f. DS18B20 Datasheet). While the conversion is taking place, the device may be polled, e.g. using in the ds.read() command in OneWire, to see if it has successfully performed a conversion.

Read Scratchpad

Once the data has been converted, it is copied into the Scratchpad memory, where it may be read. Note that the Scratchpad may be read at any time without a conversion command to recall the most previous reading, as well as the resolution of the device and other device-dependent configuration options.

Asynchronous vs. Synchronous read/write

The majority of existing code for 1Wire devices, particularly that written for Arduino, uses a very basic "Convert, Wait, Read" algorithm, even for multiple devices. This creates several problems:

Program timing for other functions

Arguably the biggest problem with using the above methodology is that unless threading measures are undertaken, the device must sit (hang) and wait for the conversion to take place if a hardcoded wait time is included. This presents a serious problem if other timed processes exist, and even if they don't -- many programs wait for user input, process data, and perform many other functions that cannot be put on hold for the time necessary for a temperature conversion process. As noted above and below, a 12-bit conversion process for a DS18B20 can take as long as 750ms. There is no reason to use the wait method, unless it is desired that the controller do nothing (at all) until the measurement conversion is complete. It is far more efficient to issue a conversion command and return later to pick up the measurement with a Read Scratchpad command once the conversion is complete.

Scaling for Poll Speed with multiple devices
Another major problem with the "Convert, Wait, Read" method is that it scales very poorly, and for no good reason. All conversion commands can be issued in series (or simultaneously, by issuing a Skip ROM and then Convert command), and the result can then be read back in succession. See discussion here: http://interfaceinnovations.org/onewireoptimization.html

Adjustment of wait time to required conversion time
The most efficient and expeditious read of 1Wire devices explicitly takes into account the conversion time of the device being read, which is typically a function of read resolution. In the example below, for example, 1000ms is given, while the datasheet lists 750ms as the maximum conversion time, and typical conversion takes place in 625ms or less. Most important, the value should be adjusted for the resolution that is currently being polled. A 9-bit conversion, for example, will take 94ms or less, and waiting for 1000ms simply doesn't make sense. As noted above, the most efficient way to poll is the use a read time slot to poll the device. In this fashion one can know exactly when the result is ready and pick it up immediately.

For discussion and code examples on this topic, please see:
http://www.cupidcontrols.com/2014/10/moteino-arduino-and-1wire-optimize-your-read-for-speed/

History

In 2007, Jim Studt created the original OneWire library that makes it easy to work with 1-Wire devices. The forum thread describes the evolution. Jim's original version only worked with arduino-007 and required a large (256-byte) lookup table to perform CRC calculations. This was later updated to work with arduino-0008 and later releases. The most recent version (unavailable at this moment?) eliminates the CRC lookup table; it has been tested under arduino-0010.

The OneWire library has a bug causing an infinite loop when using the search function. Fixes to this can be found in this Arduino thread, see posts #3, #17, #24, #27 for a variety of fixes.

Version 2.0 merges Robin James's improved search function and includes Paul Stoffregen's improved I/O routines (fixes occasional communication errors), and also has several small optimizations.

Version 2.1 adds compatibility with Arduino 1.0-beta and an improved temperature example (Paul Stoffregen), DS250x PROM example (Guillermo Lovato), chipKit compatibility (Jason Dangel), CRC16, convenience functions and DS2408 example (Glenn Trewitt).

Miles Burton derived its Dallas Temperature Control Library from it as well.

Example code

```
#include <OneWire.h>
```

```
// DS18S20 Temperature chip i/o
OneWire ds(10);  // on pin 10

void setup(void) {
  // initialize inputs/outputs
  // start serial port
  Serial.begin(9600);
}

void loop(void) {
  byte i;
  byte present = 0;
  byte data[12];
  byte addr[8];

  ds.reset_search();
  if ( !ds.search(addr)) {
     Serial.print("No more addresses.\n");
     ds.reset_search();
     return;
  }

  Serial.print("R=");
  for( i = 0; i < 8; i++) {
    Serial.print(addr[i], HEX);
    Serial.print(" ");
  }

  if ( OneWire::crc8( addr, 7) != addr[7]) {
     Serial.print("CRC is not valid!\n");
     return;
  }
```

```
if ( addr[0] == 0x10) {
   Serial.print("Device is a DS18S20 family device.\n");
}
else if ( addr[0] == 0x28) {
   Serial.print("Device is a DS18B20 family device.\n");
}
else {
   Serial.print("Device family is not recognized: 0x");
   Serial.println(addr[0],HEX);
   return;
}

ds.reset();
ds.select(addr);
ds.write(0x44,1);        // start conversion, with parasite power on at the end

delay(1000);    // maybe 750ms is enough, maybe not
// we might do a ds.depower() here, but the reset will take care of it.

present = ds.reset();
ds.select(addr);
ds.write(0xBE);        // Read Scratchpad

Serial.print("P=");
Serial.print(present,HEX);
Serial.print(" ");
for ( i = 0; i < 9; i++) {          // we need 9 bytes
  data[i] = ds.read();
  Serial.print(data[i], HEX);
  Serial.print(" ");
}
Serial.print(" CRC=");
Serial.print( OneWire::crc8( data, 8), HEX);
```

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

[Get Code]

For more compact version of the code above, as well as for description of sensor's command interface look here.

Converting HEX to something meaningful (Temperature)

In order to convert the HEX code to a temperature value, first you need to identify if you are using a DS18S20, or DS18B20 series sensor. The code to read the temperature needs to be slightly different for the DS18B20 (and DS1822), because it returns a 12-bit temperature value (0.0625 deg precision), while the DS18S20 and DS1820 return 9-bit values (0.5 deg precision).

First off, you need to define some variables, (put right under loop() above)

```
int HighByte, LowByte, TReading, SignBit, Tc_100, Whole, Fract;
```

[Get Code]

Then for a DS18B20 series you will add the following code below the `Serial.println();` above

```
LowByte = data[0];
 HighByte = data[1];
 TReading = (HighByte << 8) + LowByte;
 SignBit = TReading & 0x8000;  // test most sig bit
 if (SignBit) // negative
 {
  TReading = (TReading ^ 0xffff) + 1; // 2's comp
 }
 Tc_100 = (6 * TReading) + TReading / 4;    // multiply by (100 * 0.0625) or 6.25

 Whole = Tc_100 / 100;  // separate off the whole and fractional portions
 Fract = Tc_100 % 100;


 if (SignBit) // If its negative
 {
   Serial.print("-");
```

```
 }
 Serial.print(Whole);
 Serial.print(".");
 if (Fract < 10)
 {
   Serial.print("0");
 }
 Serial.print(Fract);

 Serial.print("\n");
```

[Get Code]

This block of code converts the temperature to deg C and prints it to the Serial output.

A Code Snippet for the DS 1820 with 0.5 Degree Resolution

Above example works only for the B-type of the DS1820. Here is a code example that works with the lower resolution DS1820 and with multiple sensors diplaying their values on a LCD. Example is working with Arduino pin 9. Feel free to change that to an appropriate pin for your use. Pin 1 and 3 of the DS1820 has to be put to ground! In the example a 5k resistor is put from pin 2 of DS1820 to Vcc (+5V). See LiquidCrystal documentation for connecting the LCD to the Arduino.

```
#include <OneWire.h>
#include <LiquidCrystal.h>
// LCD=================================================
//initialize the library with the numbers of the interface pins
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);
#define LCD_WIDTH 20
#define LCD_HEIGHT 2

/* DS18S20 Temperature chip i/o */

OneWire  ds(9);  // on pin 9
#define MAX_DS1820_SENSORS 2
byte addr[MAX_DS1820_SENSORS][8];
void setup(void)
{
```

```
 lcd.begin(LCD_WIDTH, LCD_HEIGHT,1);
 lcd.setCursor(0,0);
 lcd.print("DS1820 Test");
 if (!ds.search(addr[0]))
 {
  lcd.setCursor(0,0);
  lcd.print("No more addresses.");
  ds.reset_search();
  delay(250);
  return;
 }
 if ( !ds.search(addr[1]))
 {
  lcd.setCursor(0,0);
  lcd.print("No more addresses.");
  ds.reset_search();
  delay(250);
  return;
 }
}
int HighByte, LowByte, TReading, SignBit, Tc_100, Whole, Fract;
char buf[20];

void loop(void)
{
 byte i, sensor;
 byte present = 0;
 byte data[12];

 for (sensor=0;sensor<MAX_DS1820_SENSORS;sensor++)
 {
  if ( OneWire::crc8( addr[sensor], 7) != addr[sensor][7])
  {
```

```
  lcd.setCursor(0,0);
  lcd.print("CRC is not valid");
  return;
}

if ( addr[sensor][0] != 0x10)
{
  lcd.setCursor(0,0);
  lcd.print("Device is not a DS18S20 family device.");
  return;
}

ds.reset();
ds.select(addr[sensor]);
ds.write(0x44,1);        // start conversion, with parasite power on at the end

delay(1000);     // maybe 750ms is enough, maybe not
// we might do a ds.depower() here, but the reset will take care of it.

present = ds.reset();
ds.select(addr[sensor]);
ds.write(0xBE);         // Read Scratchpad

for ( i = 0; i < 9; i++)
{          // we need 9 bytes
  data[i] = ds.read();
}

LowByte = data[0];
HighByte = data[1];
TReading = (HighByte << 8) + LowByte;
SignBit = TReading & 0x8000;  // test most sig bit
if (SignBit) // negative
```

```
  {
    TReading = (TReading ^ 0xffff) + 1; // 2's comp
  }
  Tc_100 = (TReading*100/2);

  Whole = Tc_100 / 100;  // separate off the whole and fractional portions
  Fract = Tc_100 % 100;

  sprintf(buf, "%d:%c%d.%d\337C     ",sensor,SignBit ? '-' : '+', Whole, Fract < 10 ? 0 :
Fract);

  lcd.setCursor(0,sensor%LCD_HEIGHT);
  lcd.print(buf);
 }
}
```

RTC Library

This library allows an enables an Arduino based on SAMD architectures (es. Zero, MKRZero or MKR1000 Board) to control and use the internal RTC (Real Time Clock). A real-time clock is a clock that keeps track of the current time and that can be used in order to program actions at a certain time. Most RTCs use a crystal oscillator (like in the Arduino Zero) whose frequency is 32.768 kHz (same frequency used in quartz clocks and watches). Namely this the frequency equal to 2^15 cycles per second and so is a convenient rate to use with simple binary counter circuits. Furthermore the RTC can continue to operate in any sleep mode, so it can be used to wake up the device from sleep modes in a programmed way. Every time the board is powered, the RTC is reset and starts from a standard date. To keep the time and the RTC running it is necessary to keep the board powered. A button sized lithium battery or any battery in the 3V range, connected through a diode to the 3.3V pin, is enough to keep RTC alive if the CPU is put in sleep mode before the standard USB or VIN power is disconnected.

Functions

begin()
setMinutes()
setTime()
setMonth()
setDate()
getMinutes()
getYear()
getDay()
setAlarmMinutes()
setAlarmTime()
setAlarmMonth()
setAlarmDate()
disableAlarm()
detachInterrupt()
setHours()
setSeconds()
setYear()
setDay()
getHours()
getSeconds()
getMonth()
setAlarmHours()
setAlarmSeconds()
setAlarmYear()
setAlarmDay()
enableAlarm()
attachInterrupt()
standbyMode()

SoftwareSerial Library

The Arduino hardware has built-in support for serial communication on pins 0 and 1 (which also goes to the computer via the USB connection). The native serial support happens via a piece of hardware (built into the chip) called a UART. This hardware allows the Atmega chip to receive serial communication even while working on other tasks, as long as there room in the 64 byte serial buffer.

The SoftwareSerial library has been developed to allow serial communication on other digital pins of the Arduino, using software to replicate the functionality (hence the name "SoftwareSerial"). It is possible to have multiple software serial ports with speeds up to 115200 bps. A parameter enables inverted signaling for devices which require that protocol.

The version of SoftwareSerial included in 1.0 and later is based on the NewSoftSerial library by Mikal Hart.

Limitations

The library has the following known limitations:

If using multiple software serial ports, only one can receive data at a time.

Not all pins on the Mega and Mega 2560 support change interrupts, so only the following can be used for RX: 10, 11, 12, 13, 14, 15, 50, 51, 52, 53, A8 (62), A9 (63), A10 (64), A11 (65), A12 (66), A13 (67), A14 (68), A15 (69).

Not all pins on the Leonardo and Micro support change interrupts, so only the following can be used for RX: 8, 9, 10, 11, 14 (MISO), 15 (SCK), 16 (MOSI).

On Arduino or Genuino 101 the current maximum RX speed is 57600bps

On Arduino or Genuino 101 RX doesn't work on Pin 13
If your project requires simultaneous data flows, see Paul Stoffregen's AltSoftSerial library. AltSoftSerial overcomes a number of other issues with the core SoftwareSerial, but has it's own limitations. Refer to the AltSoftSerial site for more information.

Examples

Software Serial Example: Use this Library... because sometimes one serial port just isn't enough!

Two Port Receive: Work with multiple software serial ports.

Functions

TinyGPS library

```
/*
TinyGPS - a small GPS library for Arduino providing basic NMEA parsing
Based on work by and "distance_to" and "course_to" courtesy of Maarten Lamers.
Suggestion to add satellites(), course_to(), and cardinal(), by Matt Monson.
Precision improvements suggested by Wayne Holder.
Copyright (C) 2008-2013 Mikal Hart
All rights reserved.

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.


*/

#include "TinyGPS.h"

#define _GPRMC_TERM   "GPRMC"
#define _GPGGA_TERM   "GPGGA"

TinyGPS::TinyGPS()
  :  _time(GPS_INVALID_TIME)
  ,  _date(GPS_INVALID_DATE)
  ,  _latitude(GPS_INVALID_ANGLE)
  ,  _longitude(GPS_INVALID_ANGLE)
  ,  _altitude(GPS_INVALID_ALTITUDE)
  ,  _speed(GPS_INVALID_SPEED)
  ,  _course(GPS_INVALID_ANGLE)
```

```
  , _hdop(GPS_INVALID_HDOP)
  , _numsats(GPS_INVALID_SATELLITES)
  , _last_time_fix(GPS_INVALID_FIX_TIME)
  , _last_position_fix(GPS_INVALID_FIX_TIME)
  , _parity(0)
  , _is_checksum_term(false)
  , _sentence_type(_GPS_SENTENCE_OTHER)
  , _term_number(0)
  , _term_offset(0)
  , _gps_data_good(false)
#ifndef _GPS_NO_STATS
  , _encoded_characters(0)
  , _good_sentences(0)
  , _failed_checksum(0)
#endif
{
  _term[0] = '\0';
}

//
// public methods
//

bool TinyGPS::encode(char c)
{
  bool valid_sentence = false;

#ifndef _GPS_NO_STATS
  ++_encoded_characters;
#endif
  switch(c)
  {
  case ',': // term terminators
    _parity ^= c;
  case '\r':
  case '\n':
  case '*':
    if (_term_offset < sizeof(_term))
```

```
   {
    _term[_term_offset] = 0;
    valid_sentence = term_complete();
   }
   ++_term_number;
   _term_offset = 0;
   _is_checksum_term = c == '*';
   return valid_sentence;

  case '$': // sentence begin
   _term_number = _term_offset = 0;
   _parity = 0;
   _sentence_type = _GPS_SENTENCE_OTHER;
   _is_checksum_term = false;
   _gps_data_good = false;
   return valid_sentence;
  }

  // ordinary characters
  if (_term_offset < sizeof(_term) - 1)
   _term[_term_offset++] = c;
  if (!_is_checksum_term)
   _parity ^= c;

  return valid_sentence;
}

#ifndef _GPS_NO_STATS
void TinyGPS::stats(unsigned long *chars, unsigned short *sentences, unsigned short
*failed_cs)
{
  if (chars) *chars = _encoded_characters;
  if (sentences) *sentences = _good_sentences;
  if (failed_cs) *failed_cs = _failed_checksum;
}
#endif

//
```

```
// internal utilities
//
int TinyGPS::from_hex(char a)
{
  if (a >= 'A' && a <= 'F')
    return a - 'A' + 10;
  else if (a >= 'a' && a <= 'f')
    return a - 'a' + 10;
  else
    return a - '0';
}

unsigned long TinyGPS::parse_decimal()
{
  char *p = _term;
  bool isneg = *p == '-';
  if (isneg) ++p;
  unsigned long ret = 100UL * gpsatol(p);
  while (gpsisdigit(*p)) ++p;
  if (*p == '.')
  {
    if (gpsisdigit(p[1]))
    {
      ret += 10 * (p[1] - '0');
      if (gpsisdigit(p[2]))
        ret += p[2] - '0';
    }
  }
  return isneg ? -ret : ret;
}

// Parse a string in the form ddmm.mmmmmmmm...
unsigned long TinyGPS::parse_degrees()
{
  char *p;
  unsigned long left_of_decimal = gpsatol(_term);
  unsigned long hundred1000ths_of_minute = (left_of_decimal % 100UL) * 100000UL;
  for (p=_term; gpsisdigit(*p); ++p);
```

```
  if (*p == '.')
  {
    unsigned long mult = 10000;
    while (gpsisdigit(*++p))
    {
      hundred1000ths_of_minute += mult * (*p - '0');
      mult /= 10;
    }
  }
  return (left_of_decimal / 100) * 1000000 + (hundred1000ths_of_minute + 3) / 6;
}

#define COMBINE(sentence_type, term_number) (((unsigned)(sentence_type) << 5) |
term_number)

// Processes a just-completed term
// Returns true if new sentence has just passed checksum test and is validated
bool TinyGPS::term_complete()
{
  if (_is_checksum_term)
  {
    byte checksum = 16 * from_hex(_term[0]) + from_hex(_term[1]);
    if (checksum == _parity)
    {
      if (_gps_data_good)
      {
#ifndef _GPS_NO_STATS
        ++_good_sentences;
#endif
        _last_time_fix = _new_time_fix;
        _last_position_fix = _new_position_fix;

        switch(_sentence_type)
        {
        case _GPS_SENTENCE_GPRMC:
          _time     = _new_time;
          _date     = _new_date;
          _latitude  = _new_latitude;
```

```
        _longitude = _new_longitude;
        _speed     = _new_speed;
        _course    = _new_course;
        break;
      case _GPS_SENTENCE_GPGGA:
        _altitude  = _new_altitude;
        _time      = _new_time;
        _latitude  = _new_latitude;
        _longitude = _new_longitude;
        _numsats   = _new_numsats;
        _hdop      = _new_hdop;
        break;
      }

      return true;
    }
   }

#ifndef _GPS_NO_STATS
   else
     ++_failed_checksum;
#endif
   return false;
  }

  // the first term determines the sentence type
  if (_term_number == 0)
  {
   if (!gpsstrcmp(_term, _GPRMC_TERM))
     _sentence_type = _GPS_SENTENCE_GPRMC;
   else if (!gpsstrcmp(_term, _GPGGA_TERM))
     _sentence_type = _GPS_SENTENCE_GPGGA;
   else
     _sentence_type = _GPS_SENTENCE_OTHER;
   return false;
  }

  if (_sentence_type != _GPS_SENTENCE_OTHER && _term[0])
```

```
  switch(COMBINE(_sentence_type, _term_number))
{
  case COMBINE(_GPS_SENTENCE_GPRMC, 1): // Time in both sentences
  case COMBINE(_GPS_SENTENCE_GPGGA, 1):
    _new_time = parse_decimal();
    _new_time_fix = millis();
    break;
  case COMBINE(_GPS_SENTENCE_GPRMC, 2): // GPRMC validity
    _gps_data_good = _term[0] == 'A';
    break;
  case COMBINE(_GPS_SENTENCE_GPRMC, 3): // Latitude
  case COMBINE(_GPS_SENTENCE_GPGGA, 2):
    _new_latitude = parse_degrees();
    _new_position_fix = millis();
    break;
  case COMBINE(_GPS_SENTENCE_GPRMC, 4): // N/S
  case COMBINE(_GPS_SENTENCE_GPGGA, 3):
    if (_term[0] == 'S')
      _new_latitude = -_new_latitude;
    break;
  case COMBINE(_GPS_SENTENCE_GPRMC, 5): // Longitude
  case COMBINE(_GPS_SENTENCE_GPGGA, 4):
    _new_longitude = parse_degrees();
    break;
  case COMBINE(_GPS_SENTENCE_GPRMC, 6): // E/W
  case COMBINE(_GPS_SENTENCE_GPGGA, 5):
    if (_term[0] == 'W')
      _new_longitude = -_new_longitude;
    break;
  case COMBINE(_GPS_SENTENCE_GPRMC, 7): // Speed (GPRMC)
    _new_speed = parse_decimal();
    break;
  case COMBINE(_GPS_SENTENCE_GPRMC, 8): // Course (GPRMC)
    _new_course = parse_decimal();
    break;
  case COMBINE(_GPS_SENTENCE_GPRMC, 9): // Date (GPRMC)
    _new_date = gpsatol(_term);
    break;
```

```
    case COMBINE(_GPS_SENTENCE_GPGGA, 6): // Fix data (GPGGA)
      _gps_data_good = _term[0] > '0';
      break;
    case COMBINE(_GPS_SENTENCE_GPGGA, 7): // Satellites used (GPGGA)
      _new_numsats = (unsigned char)atoi(_term);
      break;
    case COMBINE(_GPS_SENTENCE_GPGGA, 8): // HDOP
      _new_hdop = parse_decimal();
      break;
    case COMBINE(_GPS_SENTENCE_GPGGA, 9): // Altitude (GPGGA)
      _new_altitude = parse_decimal();
      break;
  }

  return false;
}

long TinyGPS::gpsatol(const char *str)
{
  long ret = 0;
  while (gpsisdigit(*str))
    ret = 10 * ret + *str++ - '0';
  return ret;
}

int TinyGPS::gpsstrcmp(const char *str1, const char *str2)
{
  while (*str1 && *str1 == *str2)
    ++str1, ++str2;
  return *str1;
}

/* static */
float TinyGPS::distance_between (float lat1, float long1, float lat2, float long2)
{
  // returns distance in meters between two positions, both specified
  // as signed decimal-degrees latitude and longitude. Uses great-circle
  // distance computation for hypothetical sphere of radius 6372795 meters.
```

```
  // Because Earth is no exact sphere, rounding errors may be up to 0.5%.
  // Courtesy of Maarten Lamers
  float delta = radians(long1-long2);
  float sdlong = sin(delta);
  float cdlong = cos(delta);
  lat1 = radians(lat1);
  lat2 = radians(lat2);
  float slat1 = sin(lat1);
  float clat1 = cos(lat1);
  float slat2 = sin(lat2);
  float clat2 = cos(lat2);
  delta = (clat1 * slat2) - (slat1 * clat2 * cdlong);
  delta = sq(delta);
  delta += sq(clat2 * sdlong);
  delta = sqrt(delta);
  float denom = (slat1 * slat2) + (clat1 * clat2 * cdlong);
  delta = atan2(delta, denom);
  return delta * 6372795;
}

float TinyGPS::course_to (float lat1, float long1, float lat2, float long2)
{
  // returns course in degrees (North=0, West=270) from position 1 to position 2,
  // both specified as signed decimal-degrees latitude and longitude.
  // Because Earth is no exact sphere, calculated course may be off by a tiny fraction.
  // Courtesy of Maarten Lamers
  float dlon = radians(long2-long1);
  lat1 = radians(lat1);
  lat2 = radians(lat2);
  float a1 = sin(dlon) * cos(lat2);
  float a2 = sin(lat1) * cos(lat2) * cos(dlon);
  a2 = cos(lat1) * sin(lat2) - a2;
  a2 = atan2(a1, a2);
  if (a2 < 0.0)
  {
    a2 += TWO_PI;
  }
  return degrees(a2);
```

```
}

const char *TinyGPS::cardinal (float course)
{
  static const char* directions[] = {"N", "NNE", "NE", "ENE", "E", "ESE", "SE", "SSE",
"S", "SSW", "SW", "WSW", "W", "WNW", "NW", "NNW"};

  int direction = (int)((course + 11.25f) / 22.5f);
  return directions[direction % 16];
}

// lat/long in MILLIONTHs of a degree and age of fix in milliseconds
// (note: versions 12 and earlier gave this value in 100,000ths of a degree.
void TinyGPS::get_position(long *latitude, long *longitude, unsigned long *fix_age)
{
  if (latitude) *latitude = _latitude;
  if (longitude) *longitude = _longitude;
  if (fix_age) *fix_age = _last_position_fix == GPS_INVALID_FIX_TIME ?
   GPS_INVALID_AGE : millis() - _last_position_fix;
}

// date as ddmmyy, time as hhmmsscc, and age in milliseconds
void TinyGPS::get_datetime(unsigned long *date, unsigned long *time, unsigned long
*age)
{
  if (date) *date = _date;
  if (time) *time = _time;
  if (age) *age = _last_time_fix == GPS_INVALID_FIX_TIME ?
   GPS_INVALID_AGE : millis() - _last_time_fix;
}

void TinyGPS::f_get_position(float *latitude, float *longitude, unsigned long
*fix_age)
{
  long lat, lon;
  get_position(&lat, &lon, fix_age);
  *latitude = lat == GPS_INVALID_ANGLE ? GPS_INVALID_F_ANGLE : (lat / 1000000.0);
```

```
  *longitude = lat == GPS_INVALID_ANGLE ? GPS_INVALID_F_ANGLE : (lon /
1000000.0);
}

void TinyGPS::crack_datetime(int *year, byte *month, byte *day,
  byte *hour, byte *minute, byte *second, byte *hundredths, unsigned long *age)
{
  unsigned long date, time;
  get_datetime(&date, &time, age);
  if (year)
  {
    *year = date % 100;
    *year += *year > 80 ? 1900 : 2000;
  }
  if (month) *month = (date / 100) % 100;
  if (day) *day = date / 10000;
  if (hour) *hour = time / 1000000;
  if (minute) *minute = (time / 10000) % 100;
  if (second) *second = (time / 100) % 100;
  if (hundredths) *hundredths = time % 100;
}

float TinyGPS::f_altitude()
{
  return _altitude == GPS_INVALID_ALTITUDE ? GPS_INVALID_F_ALTITUDE : _altitude
/ 100.0;
}

float TinyGPS::f_course()
{
  return _course == GPS_INVALID_ANGLE ? GPS_INVALID_F_ANGLE : _course / 100.0;
}

float TinyGPS::f_speed_knots()
{
  return _speed == GPS_INVALID_SPEED ? GPS_INVALID_F_SPEED : _speed / 100.0;
}
```

```
float TinyGPS::f_speed_mph()
{
  float sk = f_speed_knots();
  return sk == GPS_INVALID_F_SPEED ? GPS_INVALID_F_SPEED :
_GPS_MPH_PER_KNOT * sk;
}

float TinyGPS::f_speed_mps()
{
  float sk = f_speed_knots();
  return sk == GPS_INVALID_F_SPEED ? GPS_INVALID_F_SPEED :
_GPS_MPS_PER_KNOT * sk;
}

float TinyGPS::f_speed_kmph()
{
  float sk = f_speed_knots();
  return sk == GPS_INVALID_F_SPEED ? GPS_INVALID_F_SPEED :
_GPS_KMPH_PER_KNOT * sk;
}

const float TinyGPS::GPS_INVALID_F_ANGLE = 1000.0;
const float TinyGPS::GPS_INVALID_F_ALTITUDE = 1000000.0;
const float TinyGPS::GPS_INVALID_F_SPEED = -1.0;
```