Arduino Workshop Manual

Koshish Foundation research Lab, Ned UET

Contents

Chapter: 1 5
Discussion Topics 5
Microcontrollers 5
Getting started with arduino 6
Install The Arduino IDE Software 7
Install the drivers for windows 7
The Integrated Development Environment (IDE) 8
Upload a sketch 9
Chapter: 2 12
Discussion Topics 12
Arduino GPIO Handling 12
Digital input & output 12
Digital Inputs 12
Code 14
Digital output 14
Code 15
Analog Input & Output 16
Prepare the Breadboard 16
Add a Potentiometer and LED 16
Program the Module 17
Code 18
Chapter: 3 19
Class Discussion Topic 19

Serial communication 19
Serial Communication Agreements 19
What Do the Serial Voltage Changes Mean? 20
Serial communication in Arduino 21
Example Program...22
Code 23
Chapter: 4 24
Class Discussion Topic 24
External Hardware Interrupts..24
External Hardware Interrupts with Arduino 25
Code 26
Special Cases...28
Chapter: 5 29
Class Discussion Topic 29
Wireless Communication & Zigbee Basics 29
How to Use X-CTU 29
Xbee Query 31
Modem Parameters and Firmware 32
XBee Interfacing with Arduino 37
Wireless doorbell 37
Components 37
Wireless doorbell connections lay-out...38
The Button Arduino 39
The Buzzer Arduino...39

Exercise 40
Chapter: 6 41
Class Discussion Topic 41
SD Card Interfacing with Arduino 41
Formatting SD card 41
Wiring 42
Arduino Library & First Test 43
Reading files 46
Reading from Files 48
Recursively listing/reading files 49
Other useful functions 50
Chapter: 7 51
Class Discussion Topic 51
Ethernet Overview 51
ENC28J60 Ethernet Module 52
ENC28J60PINOUT I/O DESCRIPTIONS 52
Initialization Steps for ENC28J60 53
Arduino SPI Interface 54
Arduino SPI Pins 55
Connect ENC28J60 with Arduino and Code 56
Ethernet Library installation 57
Example 57
Chapter: 8 59
Class discussion Topic 59

Initialization Commands..59

AT Command sequence for setting up voice call.......................................60

AT Command sequence for send sms...60

AT Command sequence for HTTP 60

AT Command sequence for TCP function 61

Chapter: 1

Discussion Topics

- Introduction to Microcontrollers
- Getting started with Arduino

Microcontrollers

The simple difference between Microprocessor and Micro controller is microcontrollers are usually designed to perform a small set of specific functions, for example as in the case of a Digital Signal Processor which performs a small set of signal processing functions, whereas microprocessors tend to be designed to perform a wider set of general purpose functions.

For example, microcontrollers are widely used in modern cars where they will each perform a dedicated task, i.e. a microcontroller to regulate the brakes on all four wheels, or a microcontroller to regulate the car air conditioning. Now, a microprocessor in a PC on other side, they perform a wide range of tasks related to the general requirements of a PC, i.e. performing the necessary calculations for a very wide set of software applications, performing I/O for the main sub-systems, peripheral control etc. This can be better understood from below notes.

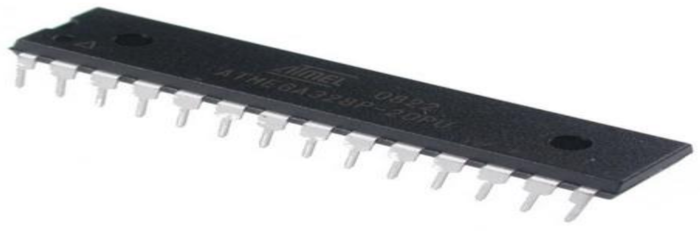
Microprocessor = cpu

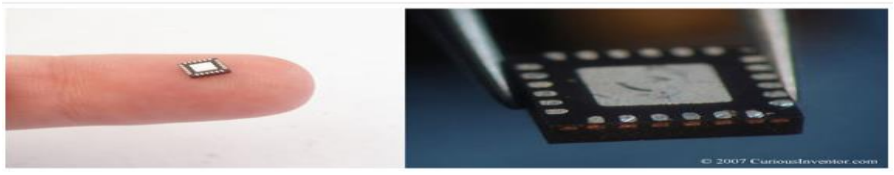
Micro controller = cpu + peripherals + memory

Peripherals = ports + clock + timers + adc converters +lcd drivers + dac + other stuff

Memory = eeprom + eprom + flash

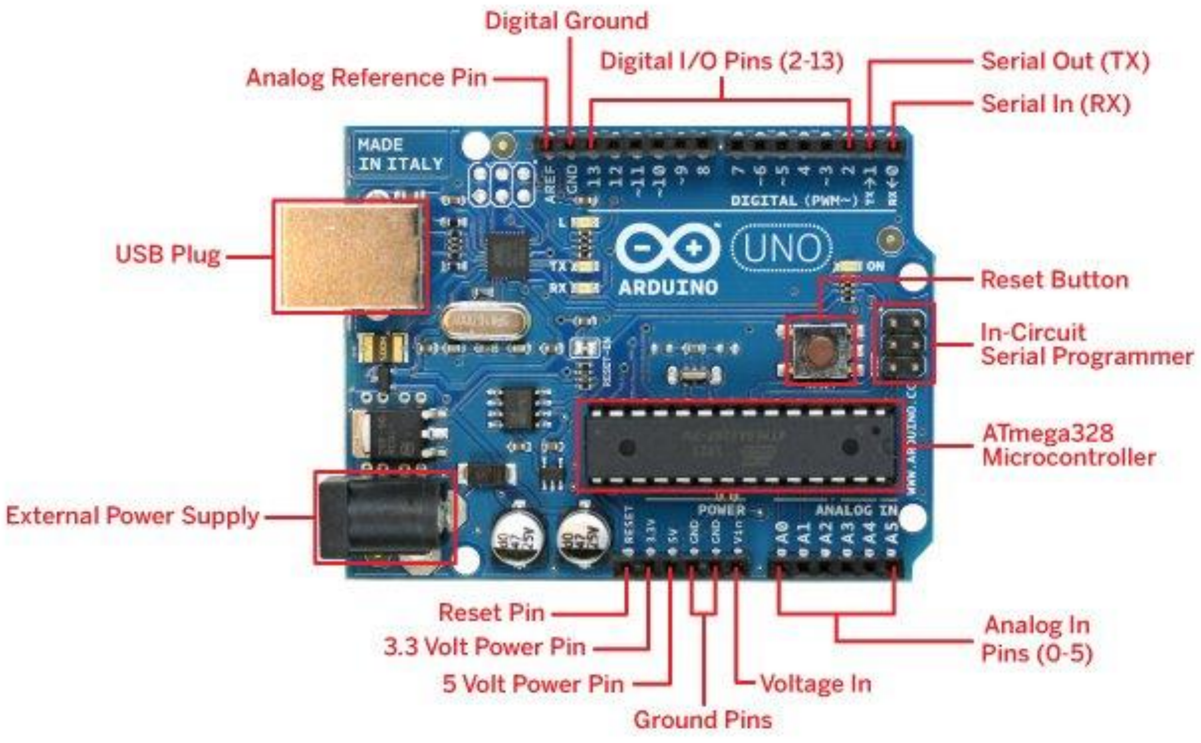
A microcontroller has a combination of all the above devices .A microprocessor is just a CPU.





Getting Started with Arduino

Arduino Uno board will be used through this course. The Arduino Uno is a microcontroller board based on the ATmega328 (datasheet). It has 14 digital



Input/output pins (of which 6 can be used as PWM outputs), 6 analog inputs, a 16 MHz ceramic resonator, a USB connection, a power jack, an ICSP header, and a reset button. It contains everything needed to support the microcontroller; simply connect it to a computer with a USB cable or power it with a AC-to-DC adapter or battery to get started. It makes easy to build and debug the projects.

Install The Arduino IDE Software

If we have access to the internet, there are step-by-step directions and the software

- Available at: http://arduino.cc/en/Main/Software

Install the Drivers for Windows

After successfully installing Arduino IDE the next step is to install the supported drivers for the cable that has been used for connecting the Arduino board to Computer. Following steps need to be followed for successful installation of the drivers.

1. Plug in board via USB and wait for Windows to begin its driver installation process. After a few moments, the process will fail. (This is not unexpected.)

2. Click on the Start Menu, and open up the Control Panel.

3. While in the Control Panel, navigate to System and Security. Next, click on System. Once the System window is up, open the Device Manager.

4. Look under Ports (COM & LPT). We should see an open port named "Arduino UNO (COMxx)".

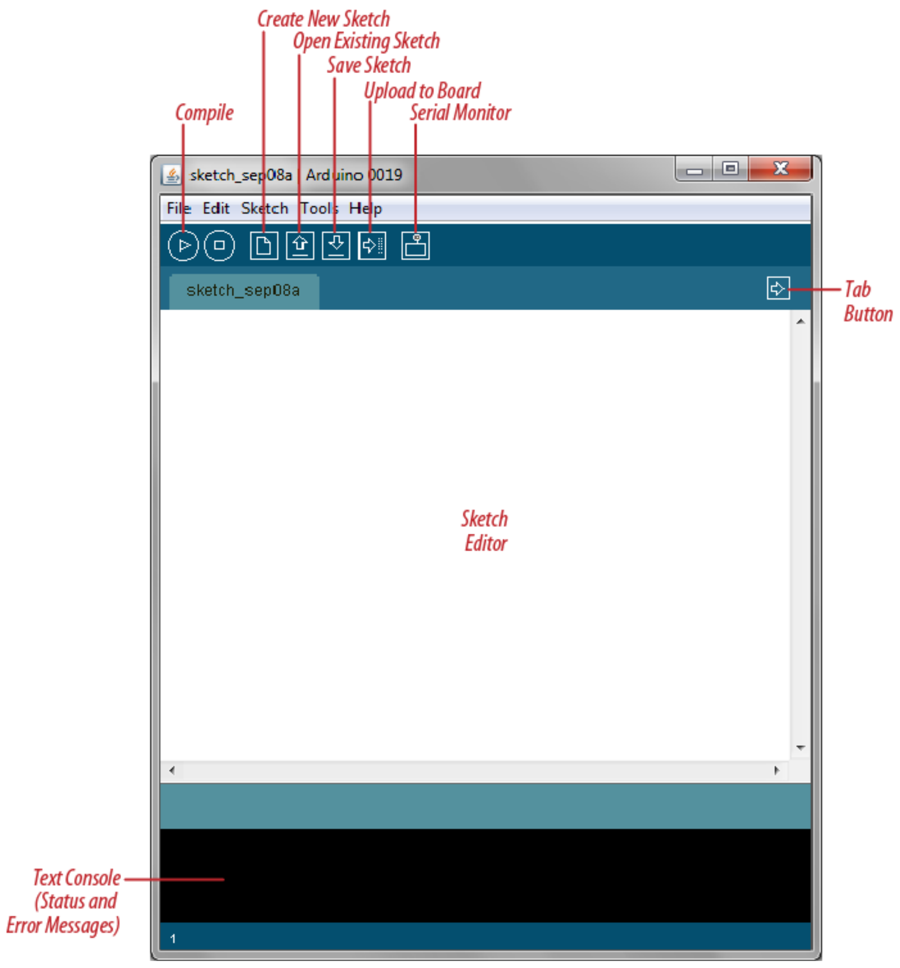
5. Right click on the "Arduino UNO (COMxx)" port and choose the "Update Driver Software" option.

6. Next, choose the "Browse my computer for Driver software" option.

7. Finally, navigate to and select the Uno's driver file, named "ArduinoUNO.inf", located in the "Drivers" folder of the Arduino Software download (not the"FTDI USB Drivers" sub-directory).

8. Windows will finish up the driver installation from there.

The Integrated Development Environment (IDE)



The Arduino IDE is used (please refer to Fig. 03) to create, open, and change sketches. Arduino calls programs "sketches" which define what the board will do. Different menus with variety of options are available in Arduino IDE which makes it easy to use. Few important and most frequently used icons are discussed as under.

• **Compile -** Before program "code" can be sent to the board, it first needs to be converted into instructions that the board understands. This process is called compiling.

• **Stop -** This stops the compilation process.

• **Create new Sketch -** This opens a new window to create a new sketch.

• **Open Existing Sketch -** This loads a sketch from a file on computer.

Save Sketch - This saves the changes to the working sketch.

• **Upload to Board** - This compiles the code and then transmit it over the USB cable to board.

• **Serial Monitor –**using this we can receive data from external devices using usb cable /DB9 connector

• **Tab Button -** This lets we create multiple files in sketch. This is for more advanced programming that is not a part of this course.

• **Sketch Editor -** This is where we write or edit sketches.

• **Text Console -** This shows what the IDE is currently doing and error messages are displayed here if there are errors in program.

• **Line Number -** This shows the line number cursor is on. It is useful since the compiler gives error messages with a line number.

Upload a Sketch

The Arduino IDE consist **few example sketches which can be** uploading **to Arduino** boards by using following steps:

1. Double-click the Arduino application.

2. Open the LED blink example sketch: File > Examples > 1.Basics > Blink

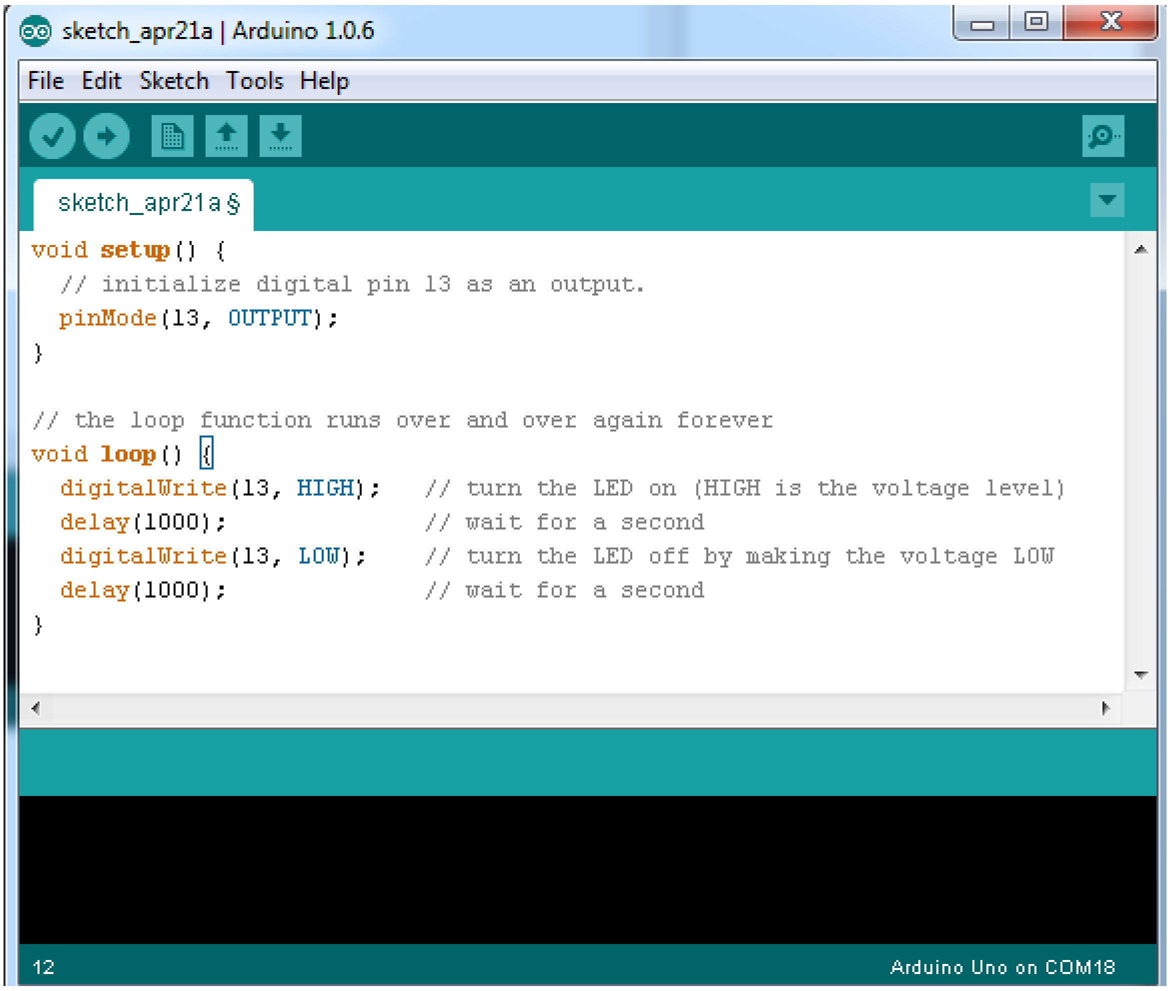
3. Select Arduino Uno under the Tools > Board menu.

4. Select serial port (if we don't know which one, disconnect the UNO and the entry that disappears is the right one.)

5. Click the Upload button.

6. After the message "Done uploading" appears, we should see the LED blinking once a second.

Here a simple sketch of blink LED is uploaded to Arduino UNO board. After successful uploading of the sketch, LED "L" on Arduino UNO board will start blinking.



```
void setup() {
  // initialize digital pin 13 as an output.
  pinMode(13, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(13, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);              // wait for a second
}
```

Now let's analyze the simple sketch for better understanding. Each program must contain at least two functions.

1. setup() which is called once when the program starts.

2. loop() which is called repetitively over and over again as long as the Arduino
.

Chapter: 2

Discussion Topics

- Arduino GPIO Handling

Arduino GPIO Handling

- Digital input & output
- Analog input & output

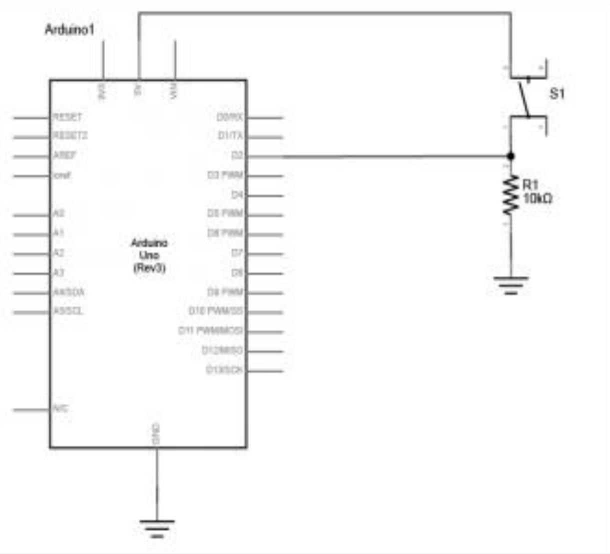
Digital input & Output

Digital input and output are the most fundamental physical connections for any microcontroller. The pins to which we connect the circuits shown in Fig. 05 are called General Purpose Input-Output (GPIO) pins.

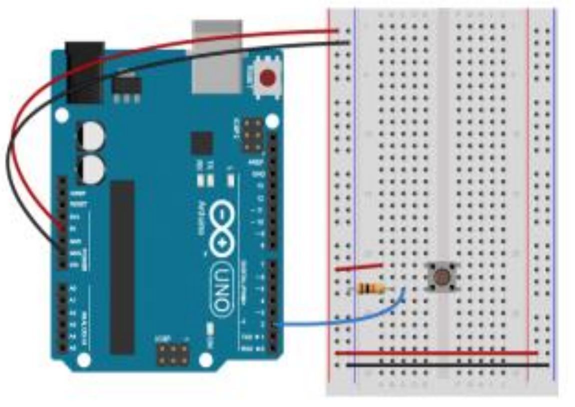
Digital Inputs

When it is needed to sense activity in the physical world using a microcontroller, the simplest activities are those in which one only need to know: Whether something is true or false. Is the vie in the room or out? Are they touching the table or not? Is the door open or closed? In these cases, we can determine what we need to know using a digital input, or switch. Digital or binary inputs to microcontrollers have two states: off and on. If voltage is flowing, the circuit is on. If it's not flowing, the circuit is off. Following figure shows Utilizing digital input pins of Arduino UNO.

Following figure shows the electrical schematic for a digital input to a microcontroller. The current has two directions it can go to ground: through the resistor or through the microcontroller. When the switch is closed, the current will follow the path of least resistance, to the microcontroller pin, and the microcontroller can then read the voltage. The microcontroller pin will



Schematic view



Arduino with switch on digital pin 2

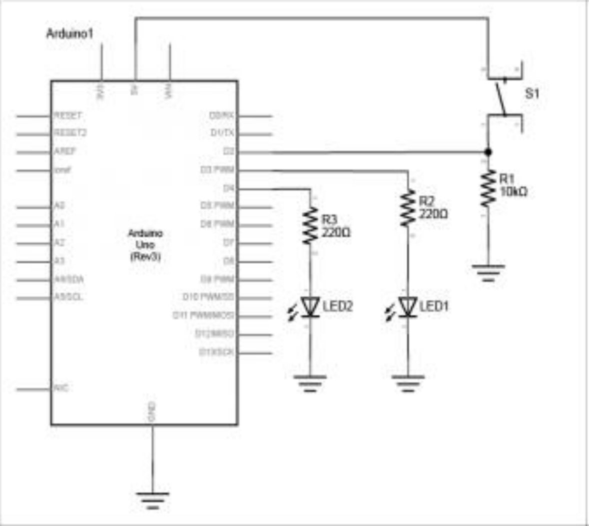
then read as high voltage or HIGH. When the switch is open, the resistor connects the digital input to ground, so that it reads as zero voltage, or LOW.

Code

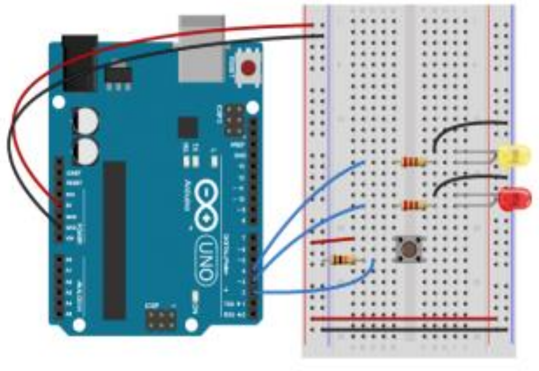
```
Void setup () {
 // declare pin 2 to be an input:
pinMode(2, INPUT);
pinMode(13, OUTPUT);
}
Void loop() {
 // read pin 2:
if (digitalRead(2) == 1) {
   // if pin 2 is HIGH, set pin 13 HIGH:
digitalWrite(3, HIGH);
 } else {
   // if pin 2 is LOW, set pin 3 LOW:
digitalWrite(3, LOW);
}
}
```

Digital Output

Just as digital inputs sense activities which have two states, digital or binary outputs allow to control activities which can have two states. With a digital output one can either turn something off or on. The digital outputs are often used to control other electrical devices besides LEDs, through transistors or relays.



Schematic view



Add LEDs on pins 3 and 4

On an Arduino module, the pin is declared as output at top of the program just like inputs. Then in the body of the program the digitalWrite() command with the values HIGH and LOW to set the pin high or low. As Shown in example code given below.

As inputs, the pins of a microcontroller can accept very little current. Likewise, as outputs, they produce very little current. The electrical signals that they read and write are mainly changes in voltage, not current. When it is needed to read an electrical change that's high-current, input current is limited using a resistor. Similarly, when it is needed to control a high-current circuit a transistor or relay can be used, both of which can control a circuit with only small voltage changes and minimal current.

Code

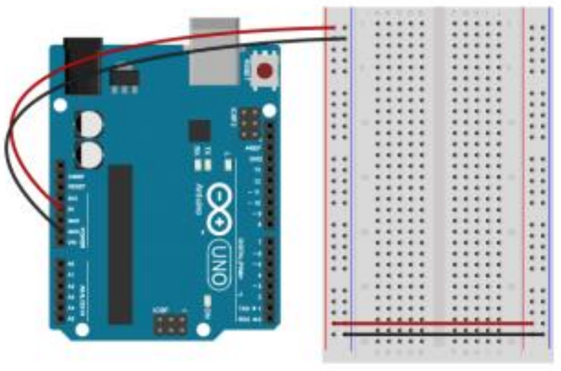
```
Void setup () {
 // declare pin 2 to be an input:
pinMode(2, INPUT);
 //declare pin 3 to be an output:
pinMode(3, OUTPUT);
pinMode(4, OUTPUT);
}
Void loop() {
 // read pin 2:
if (digitalRead(2) == 1) {
   // if pin 2 is HIGH, set pin 3 HIGH:
digitalWrite(3, HIGH);
delay(1000);
digitalWrite(3, LOW);
 } else {
   // if pin 2 is LOW, set pin 3 LOW:
digitalWrite(4, high);
delay(1000);}
}


}
```

Analog Input & Output

In this lab, we'll learn how to connect a variable resistor to a microcontroller and read it as an analog input. We'll be able to read changing conditions from the physical world and convert them to changing variables in a program. Many of the most useful sensors we might connect to a microcontroller are analog input sensors. They deliver a variable voltage, which we read on the analog input pins using the analogRead() command.

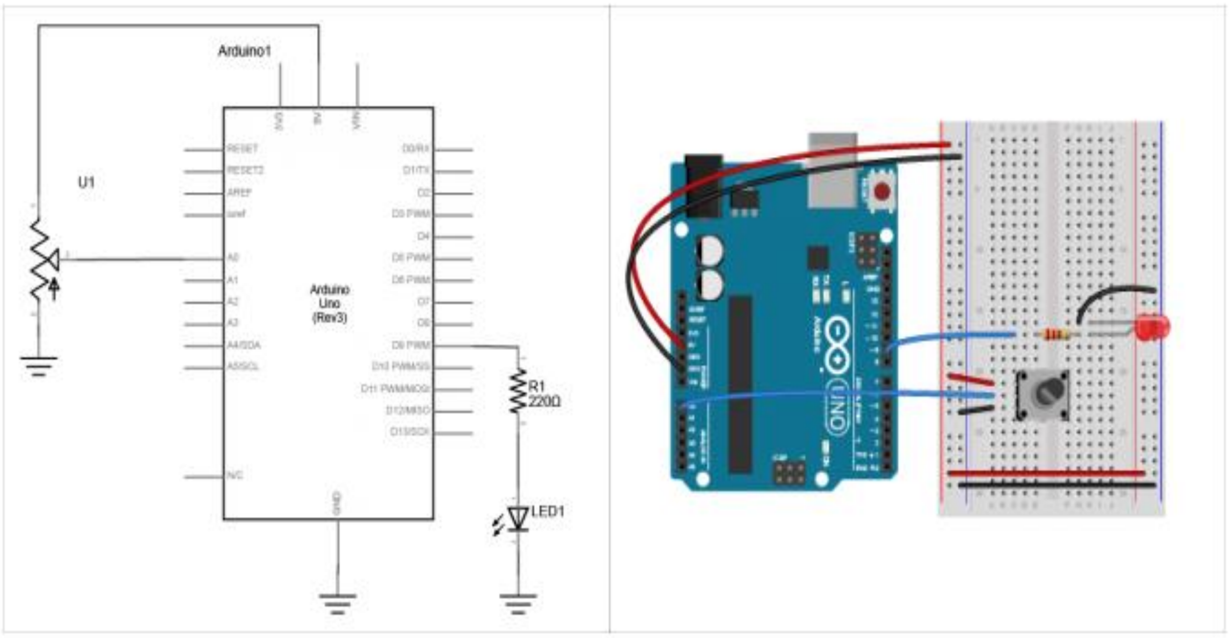
Prepare the Breadboard

Connect power and ground on the breadboard to power and ground from the microcontroller. On the Arduino module, use the 5V and any of the ground connections:



Add a Potentiometer and LED

Connect a potentiometer to analog in pin 0 of the module, and an LED and a resistor to digital pin 9:



Program the Module

Program Arduino as follows:

First, establish some global variables: one to hold the value returned by the potentiometer, and another to hold the brightness value. Make a global constant to give the LED's pin number a name.

In the setup() method, initialize serial communications at 9600 bits per second, and set the LED's pin to be an output.In the main loop, read the analog value using analogRead() and put the result into the variable that holds the analog value. Then divide the analog value by 4 to get it into a range from 0 to 255. Then use the analogWrite() command to face the LED. Then print out the brightness value. When we run this code, the LED should dim up and down as we turn the pot, and the brightness value should show up in the serial monitor.

Code

```
constintledPin = 9;      // pin that the LED is attached to

intanalogValue = 0;       // value read from the pot

int brightness = 0;        // PWM pin that the LED is on.

void setup() {

   // initialize serial communications at 9600 bps:

Serial.begin(9600);

   // declare the led pin as an output:

pinMode(ledPin, OUTPUT);

}

void loop() {

analogValue = analogRead(A0);    // read the pot value

brightness = analogValue /4;       //divide by 4 to fit in a byte

analogWrite(ledPin, brightness);     // PWM the LED with the
brightness value

Serial.println(brightness);        // print the brightness value back
to the serial monitor

}
```

Chapter: 3

Class Discussion Topic

- Serial communication
- Serial communication with Arduino

Serial Communication

In order to make two devices communicate, whether they are desktop computers, microcontrollers, or any other form of computer, we need a method of communication and an agreed-upon language. Serial communication is one of the most common forms of communication between two devices.

Serial Communication Agreements

Communicating serially involves sending a series of digital pulses back and forth between devices at a mutually agreed-upon rate. The sender sends pulses representing the data to be sent at the agreed-upon **data rate**, and the receiver listens for pulses at that same rate. This is what's known as **asynchronous serial communication**. There isn't a common clock in asynchronous serial communication; instead, both devices keep time independently, and either send or listen for new bits of data at an agreed-upon rate.

In order to communicate, the two devices need to agree on a few things:

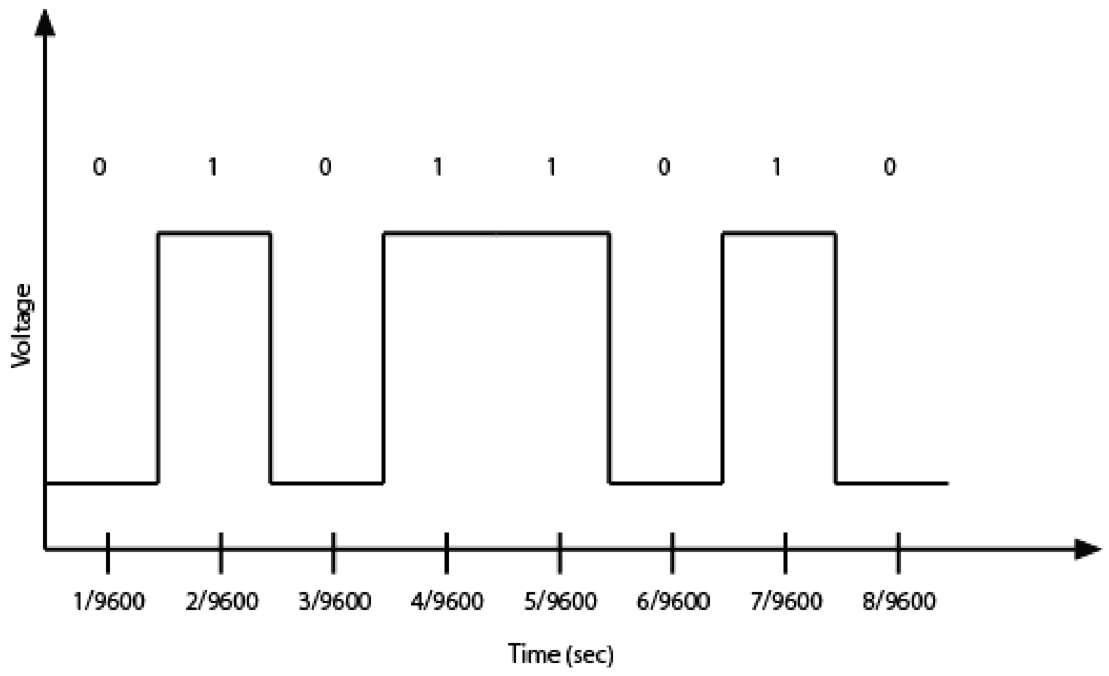
the rate at which data is sent and read the voltage levels representing a 1 or a 0 bit the meaning of those voltage levels; is a high voltage 1 and a low voltage 0, or is the signal inverted so that a low voltage is a 1 and high voltage is 0?

For example, let's say two devices are to exchange data at a rate of 9600 bits per second. First, we would make sure there's an agreed upon high and low voltage supplying each device, then we'd make three connections between the two devices:

a common ground connection, so both devices have a common reference point to measure voltage by;one wire for the sender to send data to the receiver on (transmit line for the sender);one wire for the receiver to send date to the sender on (receive line for the sender).Since the data rate is 9600 bits per second (sometimes called 9600 **baud**), the receiver will continually read the voltage on its receive wire. Every 1/9600th of a second it will interpret that voltage as a new bit of data. If the voltage is high (typically +5V or +3.3V in the case of most microcontrollers), it will interpret that bit of data as a 1. If it is low (typically 0V), it will interpret that bit of data as a 0. By interpreting the bits of data over time, the receiver can get a detailed message from the sender. At 9600 baud, for example, 1200 bytes of data can be exchanged in one second. If we have a home computer and a modem, we've seen serial communication in action. Computer's modem exchanges information with service provider's modem serially.

What Do the Serial Voltage Changes Mean?

Let's look at a byte of data being exchanged. Imagine I want to send the number 90 from one device to another. First, I have to convert the number from the decimal representation 90 to a binary representation. in binary, 90 is 01011010.



As we might guess from this diagram, both devices also have to agree on the order of the bits. Usually the sender sends the highest bit (or most significant bit) first in time, and the lowest (or least significant bit) last in time. As long as we have an agreed upon voltage, data rate, order of interpretation of bits, and agreement on what the voltage levels mean, we can exchange any data we want serially.

For the data transmission above, a high voltage indicates a bit value of 1, and a low voltage indicates a voltage of 0. This is known as true logic. Some serial protocols use inverted logic, meaning that a high voltage indicates a logic 0, and a low voltage indicates a logic 1. It’s important to know whether protocol is true or inverted. For example, RS-232, which was the standard serial protocol for most personal computers before USB came along, uses inverted logic.

Serial Communication in Arduino

Used for communication between the Arduino board and a computer or other devices. This communication happens via the Arduino board's serial or USB connection and on digitalpins 0 (RX) and 1 (TX). Thus, if we use these functions, we cannot also use pins 0 and 1 for digital I/O.

- Serial.begin(speed)
- int Serial.available()
- int Serial.read()
- Serial.flush()
- Serial.print(data)
- Serial.println(data)

Example Program

In this example we interface a potentiometer with analog pin A1 for input and digital pin2 as output. Serially transmit analog data and receive it back by connecting TX and RX. All circuit setup is described in previous examples.

Note: Processing IDE can be included for graphical data.

Code

```
int firstSensor = 0;    // first analog sensor
int inByte = 0;         // incoming serial byte
void setup()
{
  // start serial port at 9600 bps:
  Serial.begin(9600);
  while (!Serial) {
    ; // wait for serial port to connect. Needed for Leonardo only
  }
  pinMode(2, OUTPUT);   // digital sensor is on digital pin 2
  establishContact();   // send a byte to establish contact until receiver
responds
}
void loop()
{
  // if we get a valid byte, read analog ins:
  if (Serial.available() > 0) {
    // get incoming byte:
    inByte = Serial.read();
 analogWrite(2,inByte);
    // read  analog input, divide by 4 to make the range 0-255:
   firstSensor = analogRead(A1)/4;
    // send sensor values:
     while (Serial.available() <= 0){
    Serial.print (firstSensor);
 Serial.flush();
  } }  }
void establishContact() {
  while (Serial.available() <= 0) {
    Serial.print(255);   // send a 255
    delay(300);
  }
}
```

Chapter: 4

Class Discussion Topic

- External Hardware Interrupts
- External Hardware Interrupts with Arduino

External Hardware Interrupts

In this section we will explore hardware interrupts on the Arduino microcontroller. This is an electrical signal change on a microcontroller pin that causes the CPU to do the following largely in this order:

1. Finish execution of current instruction.

2. Block any further interrupts.

3. The CPU will "push" the program counter onto a "stack" or memory storage location for later retrieval.

4. The program jump to a specific memory location (interrupt vector) which is an indirect pointer to a subroutine or function.

5. The CPU loads that memory address and executes the function or subroutine that should end with a "return" instruction.

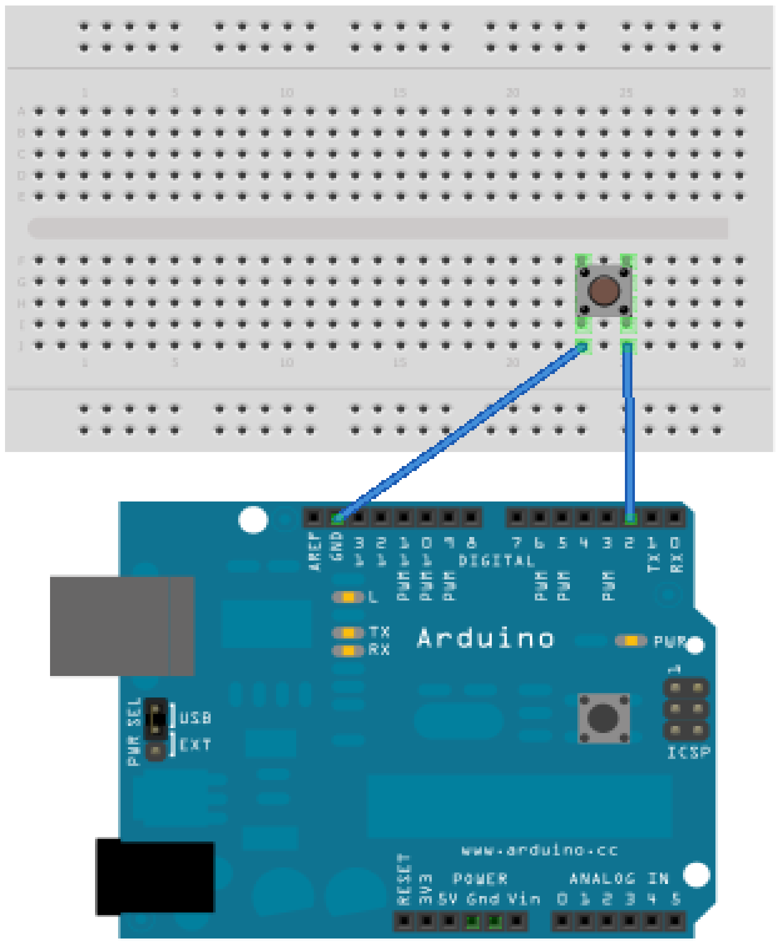
6. The CPU will "pop" the original memory address from the "stack", load it into program counter, and continue execution of original program from where the interrupt occurred.

7. Will re-enable interrupts.

The advantage of hardware interrupts is the CPU doesn't waste most of its time "polling" or constantly checking the status of an IO pin.

External Hardware Interrupts with Arduino

Most 8-bit AVR's like the ATMega328 have 2 hardware interrupts, INT0 and INT1. If we're using a standard Arduino board, these are tied to digital pins 2 and 3, respectively. Let's enable INT0 so we can detect an input change on pin 2 from a button or switch.



Code

```
#define LED 13
volatile byte state = LOW;


void setup() {
  pinMode(LED, OUTPUT);
  attachInterrupt(0, toggle, RISING);
}


void loop() {
  digitalWrite(LED, state);
}


void toggle() {
  state = !state;
}
```

Above is our first sample program. The label "LED" to the compiler is defined as the number 9. So digitalWrite(LED, HIGH) is the same as digitalWrite(13, HIGH) which is the same as digitalWrite(LED, 1), etc.

The second line defines the variable "state" as both a byte (8-bit variable) and "volatile". This differs from the usual byte, integer, or float in how the system and compiler use it. If being used as an interrupt routine variable be it byte, float, etc. is must have a "volatile" declaration. The variable is set to 0 - in reality we are using only one of the eight bits bit 0.

In setup() we come to the function **attachInterrupt(interrupt, function, mode)** where "interrupt" is the interrupt number 0 to 5; "function" is known as the interrupt service routine or ISR a function address pointed to by the interrupt vector location; "mode" configures the hardware electrical characteristics for an interrupt. This is done internally by the compiler and hidden from the user.

So in this case of attachInterrupt(0, toggle, FALLING) zero corresponds to interrupt 0 on DP2, toggle() is the ISR routine at the bottom of the program, and RISING means when an electrical signal goes from 0V to 5V the toggle() ISR performs its function - what started as state = LOW is now state = HIGH and vise-versa. The right "}" is considered the "return" command - in fact "return;" placed after "state = !state;" is ignored and won't produce an error.

In other words loop() will simply keep writing the variable "state" to the LED on DP9, and on an interrupt caused by pressing SW0 will halt, save address counter, jump to ISR toggle(), will come back where it stopped and continue.

Assuming the LED1 is off press SW0 and the LED will come on, release nothing happens. Press again and the LED is off. What toggle() really does is a bitwise XOR of 1; state = !state is the same as state = state ^ 1.

With attachInterrupt(interrupt, function, mode) there are four "mode" declarations that defines when the interrupt should be triggered and are predefined as:

RISING to trigger when the pin goes from low to high - as wired above press switch and LED state will toggle.

FALLING for when the pin goes from high to low - press switch nothing, release switch LED state will toggle.

CHANGE to trigger the interrupt whenever the pin changes value - press switch LED state will toggle, release switch LED will toggle again.

LOW to trigger the interrupt whenever the pin is low - sort of like FALLING but erratic - don't use.

Special Cases

It’s possible to reassign interrupts using the **attachInterrupt()** function again, but its also possible to remove them by calling the function **detachInterrupt().** If there is a section of code that is time sensitive, and it's important that an interrupt is not called during that time frame, we can temporarily disable interrupts with the function **noInterrupts()** and then turn them back on again afterward with the function interrupts(). This definitely comes in handy on occasion.

Chapter: 5

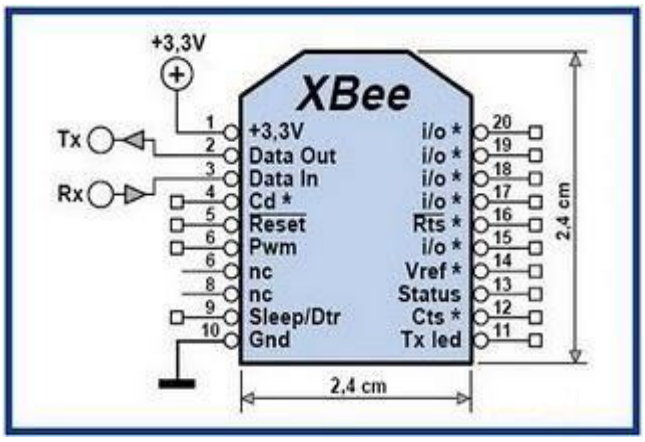
Class Discussion Topic

- Wireless Communication & Zigbee Basics
- Zigbee Modules Interfacing with Arduino

Wireless Communication & Zigbee Basics

The technology defined by the ZigBee specification is intended to be simpler and less expensive than other WPANs, such as Bluetooth. ZigBee is targeted at radio-frequency (RF) applications that require a low data rate, long battery life, and secure networking.

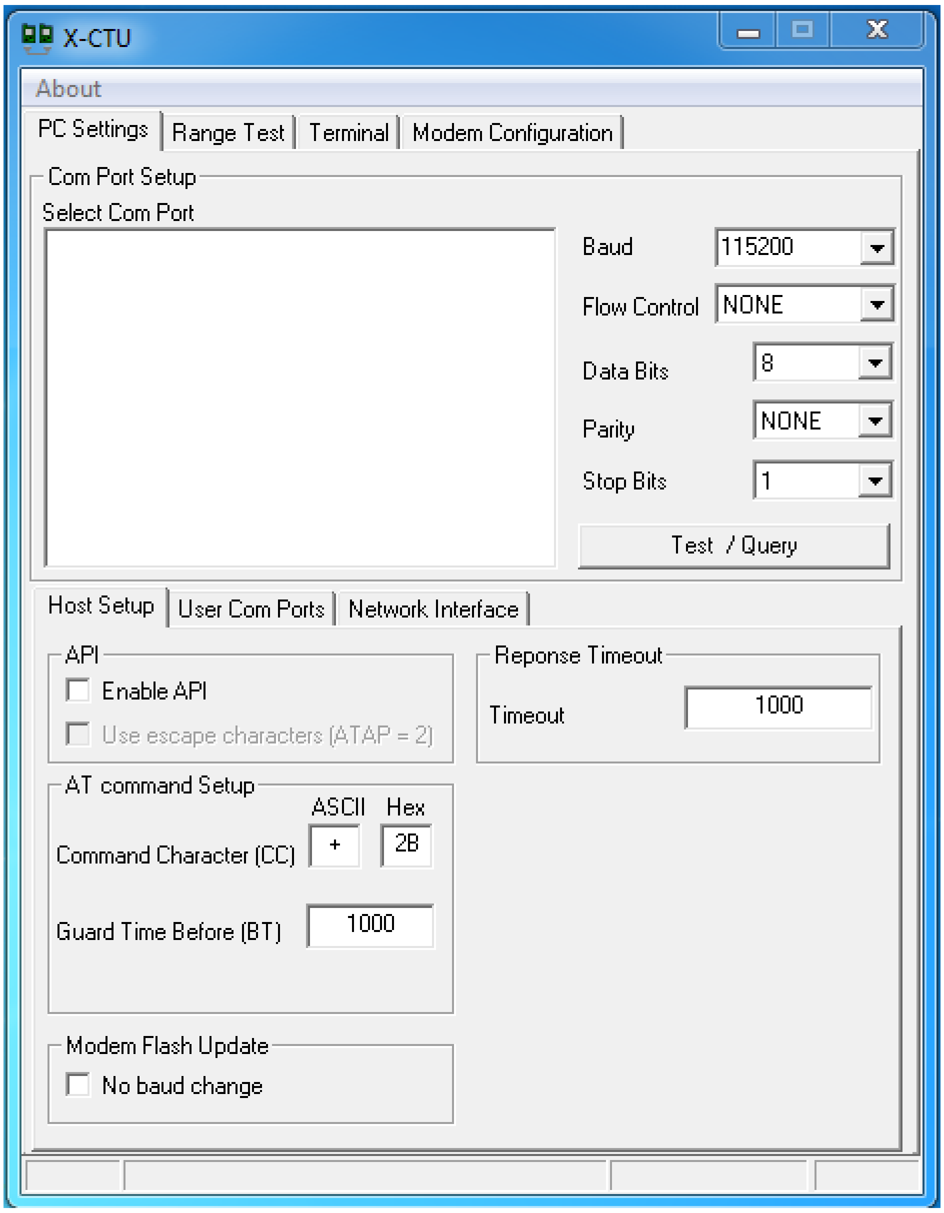
ZigBee is a low-cost, low-power, wireless mesh networking proprietary standard. The low cost allows the technology to be widely deployed in wireless control and monitoring applications, the low power-usage allows longer life with smaller batteries, and the mesh networking provides high reliability and larger range.



How to Use X-CTU

Digi International offers convenient tools for Xbee module programming - X-CTU. With this software, the user is able to upgrade the firmware, update the parameters, perform communication testing easily.

Before we can talk to an XBee, except USB cable we will need to get an USB adapter for XBee. With the USB adapter, we can communicate with Xbee through "USB Serial Port". We may have more than one device on serial port, for example, we want to test the wireless communication and connect two XBee to PC. So we will more devices on serial port. In either case, select the correct one that we want to perform operations.


X-CTU

About

PC Settings | Range Test | Terminal | Modem Configuration

Com Port Setup

Select Com Port

Baud: 115200

Flow Control: NONE

Data Bits: 8

Parity: NONE

Stop Bits: 1

Test / Query

Host Setup | User Com Ports | Network Interface

API

☐ Enable API

☐ Use escape characters (ATAP = 2)

Reponse Timeout

Timeout: 1000

AT command Setup

	ASCII	Hex
Command Character (CC)	+	2B
Guard Time Before (BT)	1000	

Modem Flash Update

☐ No baud change

Xbee Query

This is an easy way to test and check if an XBee is working and configuring properly. After parameter modification and firmware upgrading, we need to do this for checking if everything is ok. We will have a very little chance to get a problem Xbee. If we got problem to Test / Query an XBee, usually it is due to the wrong parameter setting.

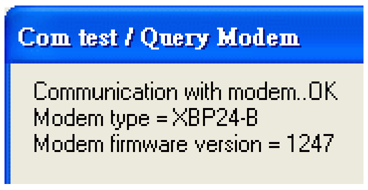
For a successful two way communication, the most primary principle is the "Baud Rate" should match each other.

For a new Xbee module , follow this procedure to query.

[1] Select com port in section : "PC Settings"
[2] NONE
[3] Stop Bits : 1
[4] Enable API : Uncheck
[5] Baud Rate set to 9600 (only for the new Xbee module , set the value to it.
[6] Flow Control : NONE
[7] Data Bits : 8
[8] Parity : none

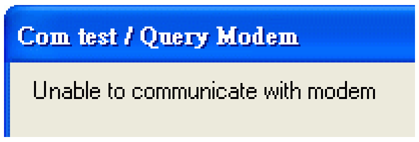
1. Click "Test / Query"

If we had set the Baud Rate to other value, then we should change to it.



```
Communication with modem..OK
Modem type = XBP24-B
Modem firmware version = 1247
```

When everything comes to the right place, this window will appear. If we see any other kind of message window, it means something wrong even we see an OK on it. It is not ok without this message and window. With the same setting, if we only got this occasionally, it means the communication is unstable. We may need to use an XBee adapter with a better quality, or try another XBee. The modem type and firmware version means the firmware be programmed in this XBee. It is possible to have other types of firmware, depends on usage for XBee.



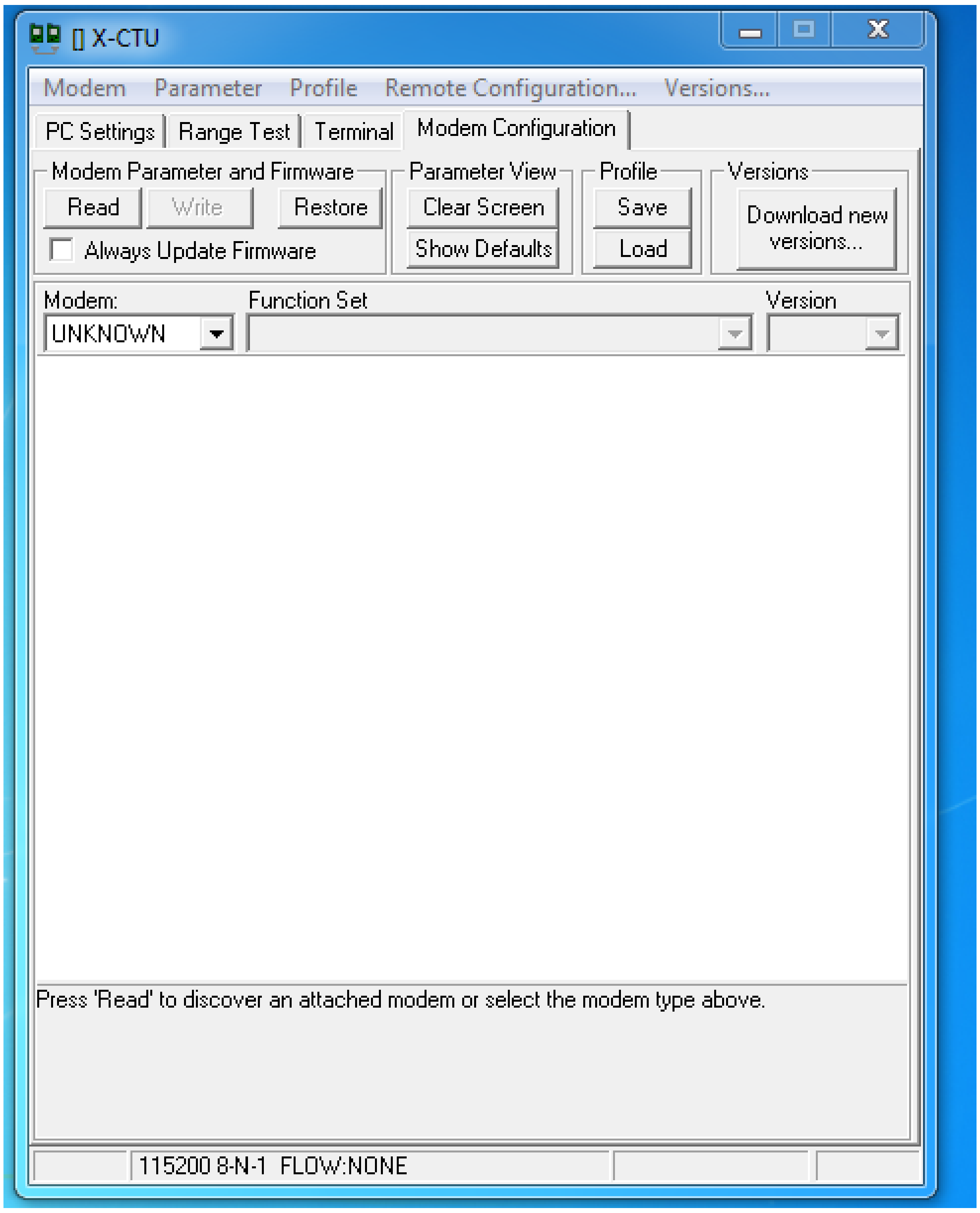
Com test / Query Modem

Unable to communicate with modem

If something goes wrong, then we will get this window. For most cases, it can be solved by just changing the Baud Rate and un/check API Mode.

Modem Parameters and Firmware

After connect to an XBee , we can "Read" the parameters include Baud Rate and the firmware information (Modem Type , version , and its Function Sets for different purposes) from here. We can also "Write" parameter and firmware into the connected XBee after done all the setting. We don't need to check off "Always update firmware" if we just want to try different parameters for XBee. One more thing is, if we cracked XBee by updating the wrong firmware, the "Restore" button can not restore it.



Press 'Read' to discover an attached modem or select the modem type above.

115200 8-N-1 FLOW:NONE

Versions

"Download new versions..." downloads the latest version for all the modem types from internet. It takes quite a while for updating the firmware database.

Parameter View

"Clear Screen" clean the information in parameter window. "Show Defaults" shows the default parameters of a modem type and its function set.

Profile

it is useful when we want to try a different parameter setting for XBee. We can load a saved profile for XBee after we found problems with the new one. Except all the parameters, it also stores the information of modem type, function set, and its version.

Modem, Function Set, and Version

Pick up the right modem type which XBee belongs to , then select the proper function set for particular usage. We may also pick up the older version for a modem type. If we pick up the wrong one and write it to the XBee, then the XBee might not response to Test / Query in the PC Settings tab.

Modify Xbee Parameter

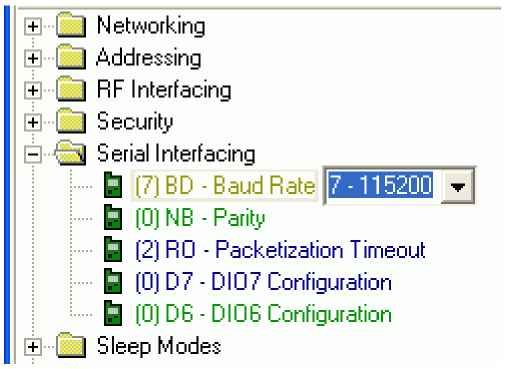
For optimizing XBee , we will need to modify the parameters in there. The most basic and important way for optimization is to modify its Baud Rate. We can boost the transfer rate from 9600 bps to 115200 immediately. Base on the datasheet, the Baud Rate of an XBee for a serial interface can be set up to 115200 bps, while the RF Data Rate is 250,000 bps. This means, it is safe to use the highest serial interface transfer rate for XBee, unless the device which connects to it cannot afford this transfer rate. After the modification, remember to reconfigure the PC Settings to the updated Baud Rate. Since the parameter of XBee we connected, has changed by this operation.

For change the communication baud rate of a new Xbee module, follow this procedure:

1. Execute [Xbee Query] and get a modem type and firmware version
2. Click on TAB : "Modem Configuration"
3. Uncheck : "Always update firmware"

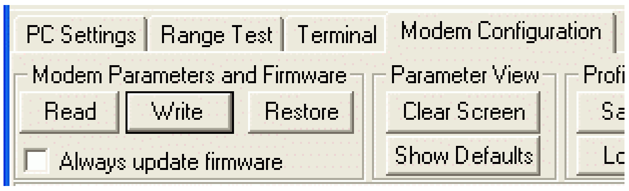
4. Click on "Read" , it will read and display all the parameters of the Xbee Module if success
5. Searching for "Serial Interfacing" > "BD - Baud Rate" in the parameters
6. Change Baud Rate to "7 - 115200"
7. Click on "Write" , it'll show the status of writing parameter if success
8. Click on TAB : "PC Settings"
9. Set Baud to 115200
10. Click on "Test / Query"

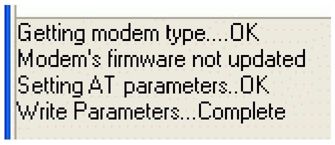
On step 3. , since we just want to modify the parameter not update the firmware, so uncheck it. It is kind of annoying not necessary to update the firmware all the time, unless we know what it is for.


Networking
Addressing
RF Interfacing
Security
Serial Interfacing
(7) BD - Baud Rate 7 - 115200
(0) NB - Parity
(2) RO - Packetization Timeout
(0) D7 - DIO7 Configuration
(0) D6 - DIO6 Configuration
Sleep Modes

There're a lot of parameters in here, now just searching for "Serial Interface" and "BD - Baud Rate". Then change it to "7 - 115200".

On step 7. , click on "Write" button to start updating the parameter. Make sure we did not check the option "Always update firmware".





```
Getting modem type....OK
Modem's firmware not updated
Setting AT parameters..OK
Write Parameters...Complete
```

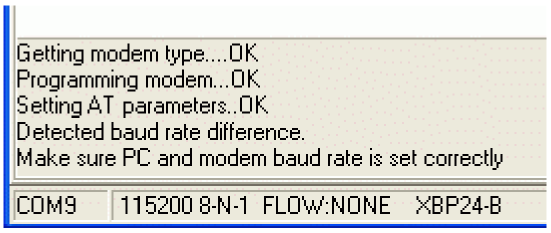
We will get this information if everything is right. It tells we it didn't update the firmware and just configure the AT parameters for we. Remember go back to PC Settings and set the new Baud Rate , else we won't be able to talk to XBee.

Update Xbee Firmware

In X-CTU , click on TAB "Modem Configuration" and we will get this. The user interface for firmware downloading and upgrading. Before we rush into update the firmware and crack the XBee , we may want to have a look about what does these sub-sections and buttons for. So we don't have to trying errors, waste time, and crack the XBee by accident.

The procedure for upgrading firmware

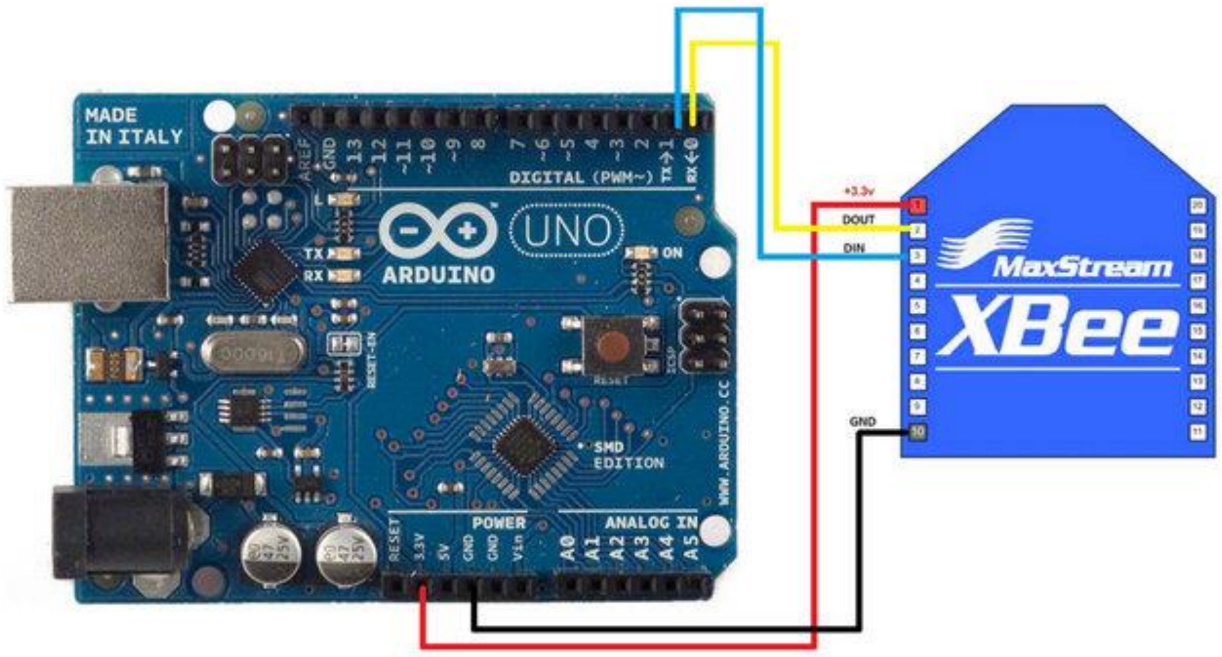
1. Setup a correct "PC Settings"
2. Click on TAB "Modem Configuration"
3. Click "Download new versions" from web
4. Wait for auto downloading complete (30 mins)
5. Have a coffee , hot tea or something.
6. Select a correct Modem Type
7. Select a proper Function Set Check : "Always update firmware"
8. Click "Write"
9. The nice programming status bar will show up when updating the firmware



```
Getting modem type....OK
Programming modem...OK
Setting AT parameters..OK
Detected baud rate difference.
Make sure PC and modem baud rate is set correctly
COM9    115200 8-N-1  FLOW:NONE    XBP24-B
```

After done , we should see this message. It also reminds we the baud rate setting of PC and XBee should match each other for Test / Query. If everything is right, we should be able to Test / Query the XBee. We may need to reboot the XBee manually to make it response to the Test / Query.

XBee Interfacing with Arduino



Wireless Doorbell

In this example we will learn to make a wire-less doorbell system. It is composed by two components: the switch and the buzzer as shown in connection lawet figure. When we press a switch a buzzer will sound.

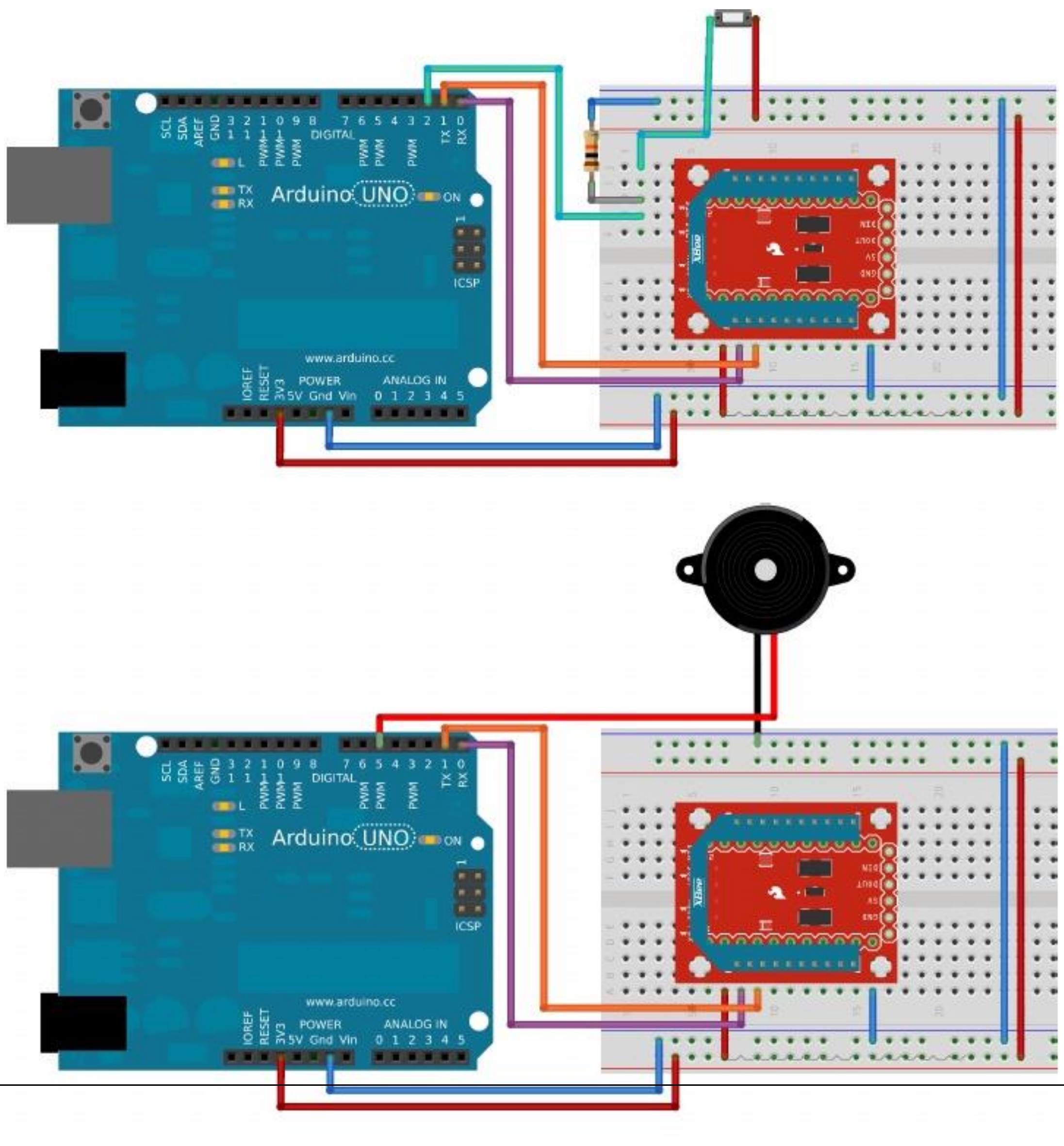
Components

Even though the two components may fit in one bread-board; to make it more real, it is better to use two separate breadboards.

- Hookup wires. It is recommended to have at least four different colors.
- Two Arduino boards.
- USB cable for the Arduinos.
- One 10KΩ resistor.
- One momentary switch or pushbutton for input.
- One buzzer for output.
- One XBee radio configured as ZIGBEE COORDINATOR AT
- One XBee radio configured as ZIGBEE ROUTER AT

- USB cable for the XBee breakout board. Every ZigBee network has only one coordinator. Other nodes can be configured as routers. It is strongly suggested that we mark down the XBees to distinguish the coordinator from the router(s).

Wireless Doorbell Connections Lay-out



The Button Arduino

This code corresponds with the button side of the example. There’s a trap! When uploading programs to Arduino, disconnect the digital pin 0 (RX) and then reconnect it after the loading is completed. Otherwise we will receive an error.

```
int BUTTON = 2;
void setup() {
    pinMode(BUTTON, INPUT);
    Serial.begin(9600);
}
void loop() {
// send a capital D over the serial port if the
   button is pressed
    if (digitalRead(BUTTON) == HIGH) {
        Serial.print('D');
        delay(10); // prevents overwhelming the
           serial port
    }
}
```

The Buzzer Arduino

This Arduino will receive a signal when the button is pressed and will ring

```
int BELL = 5;
void setup() {
    pinMode(BELL, OUTPUT);
    Serial.begin(9600);
}
void loop() {
    // look for a capital D over the serial port and
        ring the bell if found
            analogWrite(BELL, 180); //ranging from
                0-255
            delay(10);
            analogWrite(BELL, 0);
        }
    }
}
```

the buzzer.

Exercise

The Arduino on the buzzer side can send a confirmation to the Arduino on the push-button side. This confirmation means that the initial message has been received and the buzzer is buzzing. Include an additional LED on the switch side that lights up when this confirmation is received. Now try the doorbell again. Confirm that the LED lights up, when the button is pushed. Finally, disconnect the Arduino on the buzzer side and try to push the button again. There should be no sound and no light.

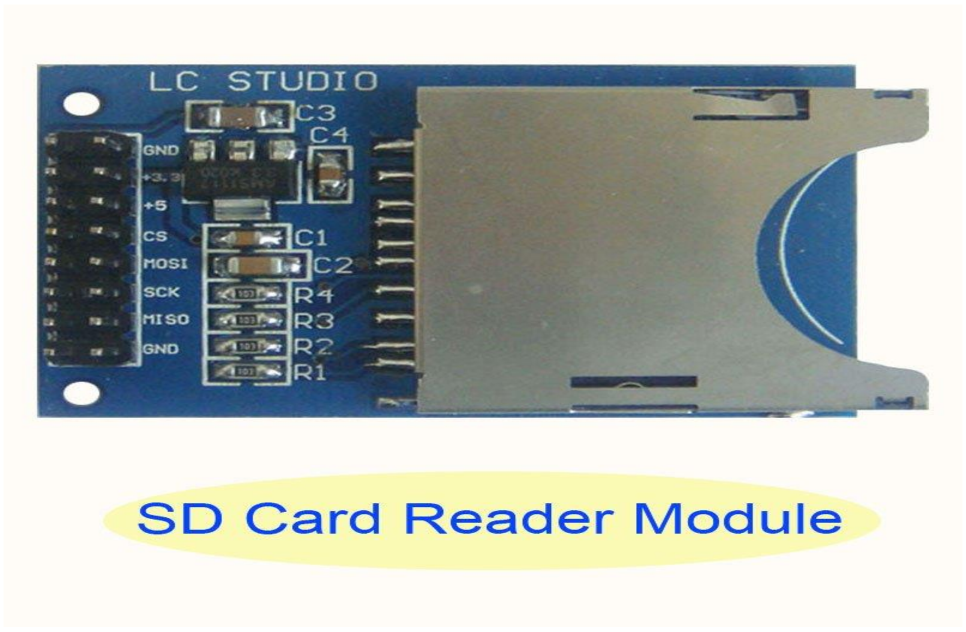
Chapter: 6

Class Discussion Topic

- SD Card Interfacing with Arduino

When we are doing any sort of data logging, graphics or audio, we'll need at least a megabyte of storage, and 64 M is probably the minimum. To get that kind of storage we're going to use the same type that's in every digital camera and mp3 player: flash cards! Often called SD or microSD cards, they can pack gigabytes into a space smaller than a coin.

SD Card Interfacing with Arduino



Formatting SD Card

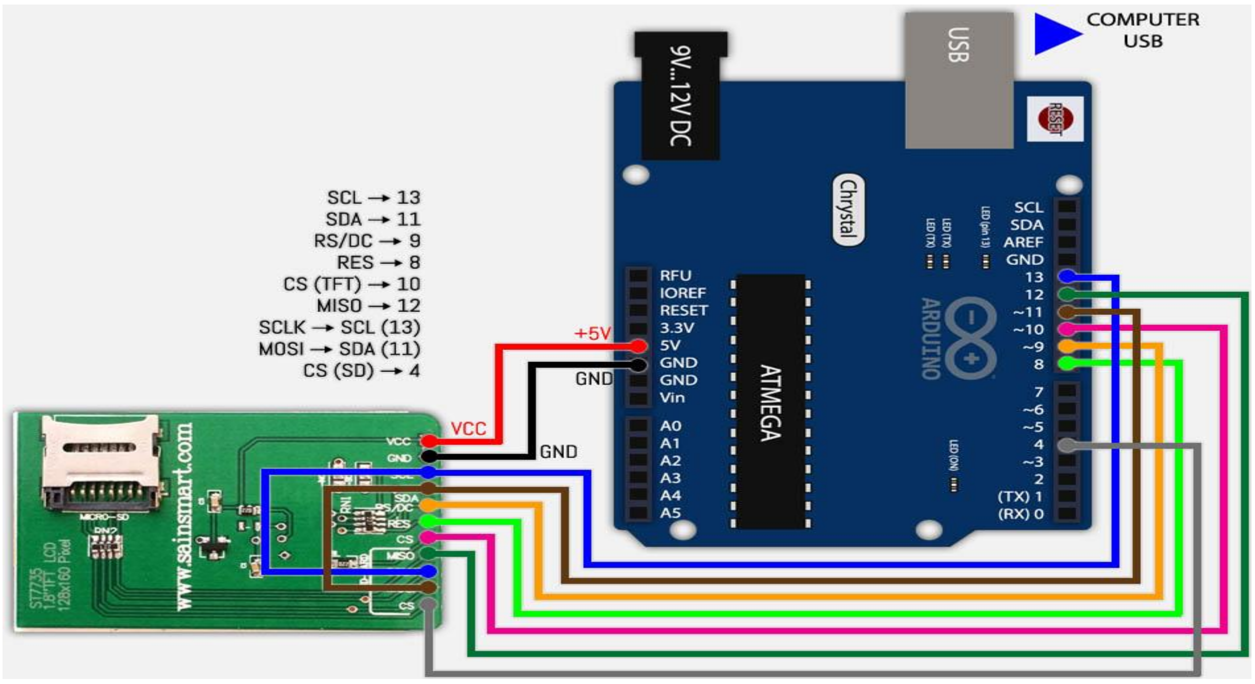
Even though we can/could use SD card 'raw' - it's most convenient to format the cardto a filesystem. For the Arduino library we'll be discussing, and nearly every other SD library,the card must be formatted FAT16 or FAT32. Some only allow one or the other. The ArduinoSD library can use either.

Wiring

Now that card is ready to use, we can wire up the microSD breakout board.it uses Arduino SPI interface.

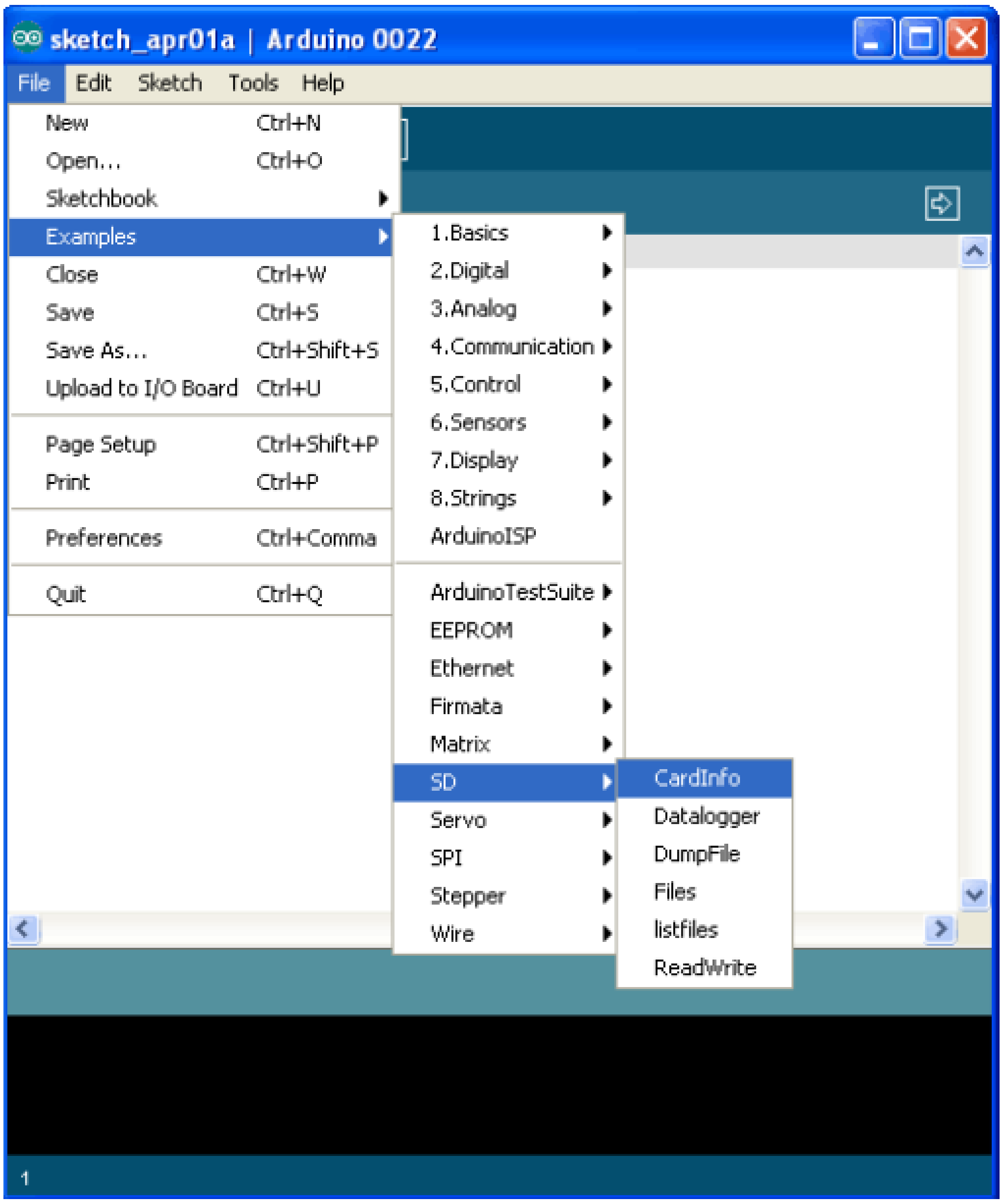
For Arduinos such as the Duemilanove/Diecimila/Uno those pins are digital 13 (SCK), 12(MISO) and 11 (MOSI). We will also need a fourth pin for the 'chip/slave select' (SS) line. Traditionally this is pin 10 but we can actually use any pin we like. If we have a Mega, the pins are different! We'll want to use digital 50 (MISO), 51 (MOSI), 52 (SCK), and for theCS line, the most common pin is 53 (SS).

- Connect the 5V pin to the 5V pin on the Arduino
- Connect the GND pin to the GND pin on the Arduino
- Connect CLK to pin 13 or 52
- Connect DO to pin 12 or 50
- Connect DI to pin 11 or 51
- Connect CS to pin 10 or 53



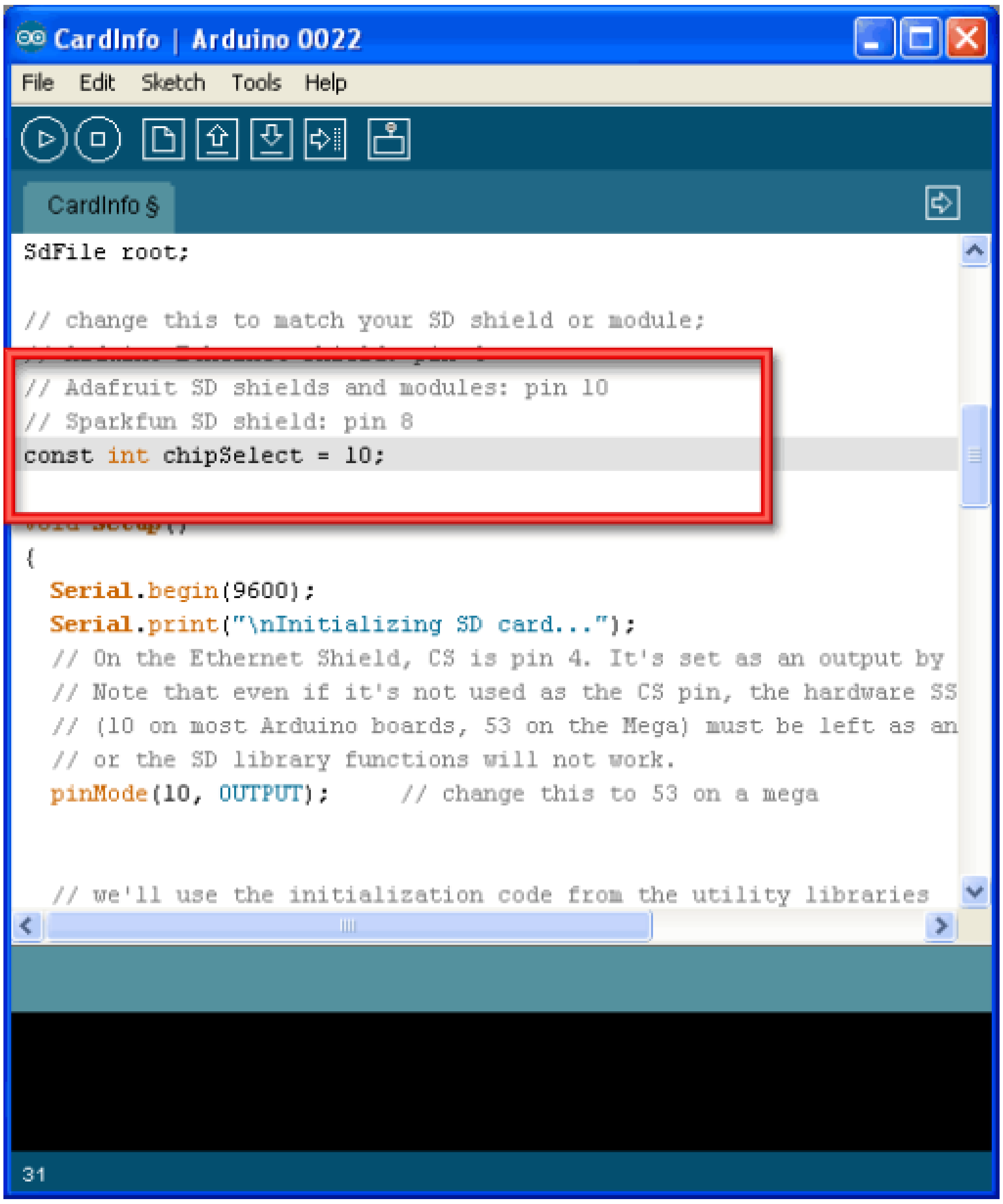
Arduino Library & First Test

- Interfacing with an SD card is a bunch of work, but luckily for us, Adafruit customer fat16lib(William G) has written a very nice Arduino library just for this purpose and it's now part of the Arduino IDE known as SD . We can see it in the Examples submenu.
- Next, select the **CardInfo** example sketch.



- This sketch will not write any data to the card, just tells 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!

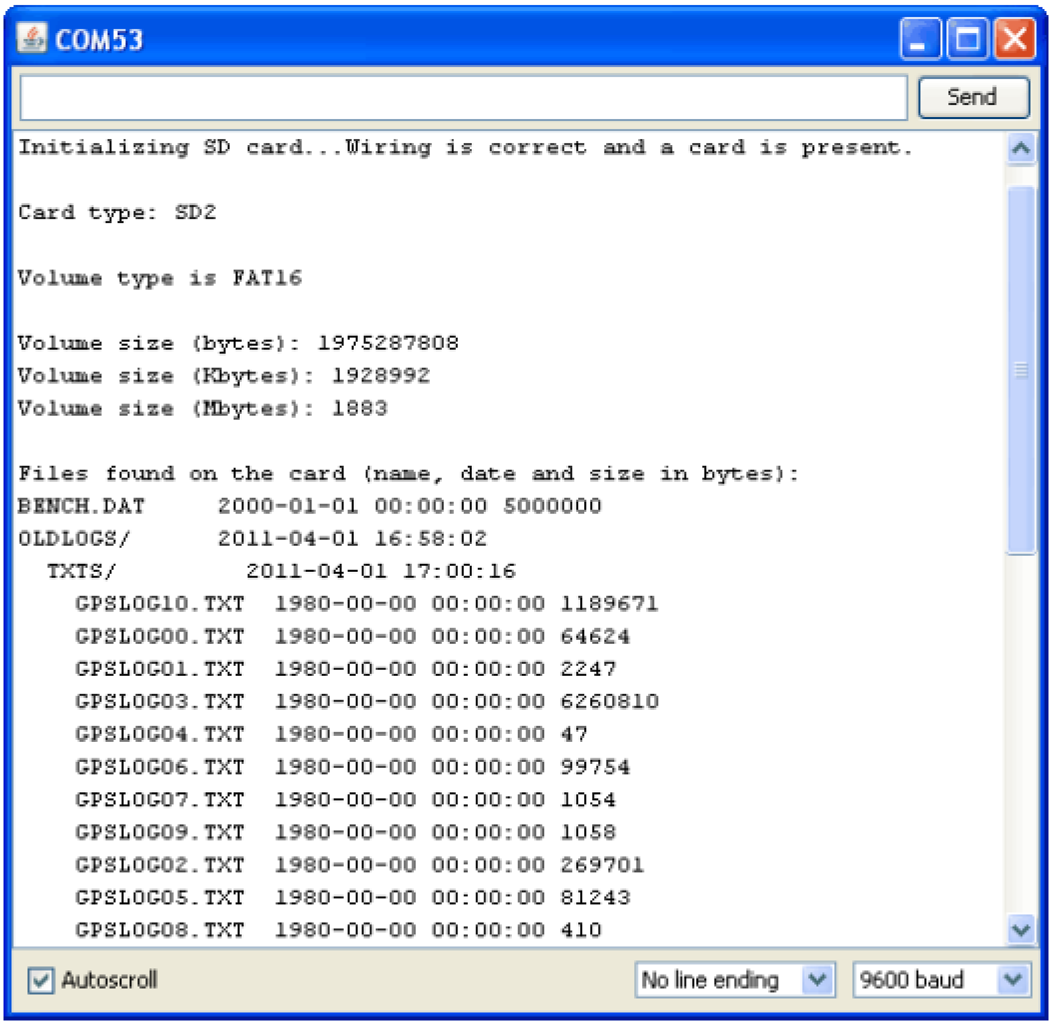
- Go to the beginning of the sketch and make sure that the chipSelect line is correct, for this wiring we're using digital pin 10 so change it to 10!



```
SdFile root;

// change this to match your SD shield or module;
// Arduino Ethernet shield: pin 4
// Adafruit SD shields and modules: pin 10
// Sparkfun SD shield: pin 8
const int chipSelect = 10;

void setup()
{
  Serial.begin(9600);
  Serial.print("\nInitializing SD card...");
  // On the Ethernet Shield, CS is pin 4. It's set as an output by
  // Note that even if it's not used as the CS pin, the hardware SS
  // (10 on most Arduino boards, 53 on the Mega) must be left as an
  // or the SD library functions will not work.
  pinMode(10, OUTPUT);     // change this to 53 on a mega


  // we'll use the initialization code from the utility libraries
```

- OK, now insert the SD card into the breakout board and upload the sketch.

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



```
Initializing SD card...Wiring is correct and a card is present.

Card type: SD2

Volume type is FAT16

Volume size (bytes): 1975287808
Volume size (Kbytes): 1928992
Volume size (Mbytes): 1883

Files found on the card (name, date and size in bytes):
BENCH.DAT      2000-01-01 00:00:00 5000000
OLDLOGS/       2011-04-01 16:58:02
  TXTS/          2011-04-01 17:00:16
    GPSLOG10.TXT  1980-00-00 00:00:00 1189671
    GPSLOG00.TXT  1980-00-00 00:00:00 64624
    GPSLOG01.TXT  1980-00-00 00:00:00 2247
    GPSLOG03.TXT  1980-00-00 00:00:00 6260810
    GPSLOG04.TXT  1980-00-00 00:00:00 47
    GPSLOG06.TXT  1980-00-00 00:00:00 99754
    GPSLOG07.TXT  1980-00-00 00:00:00 1054
    GPSLOG09.TXT  1980-00-00 00:00:00 1058
    GPSLOG02.TXT  1980-00-00 00:00:00 269701
    GPSLOG05.TXT  1980-00-00 00:00:00 81243
    GPSLOG08.TXT  1980-00-00 00:00:00 410
```

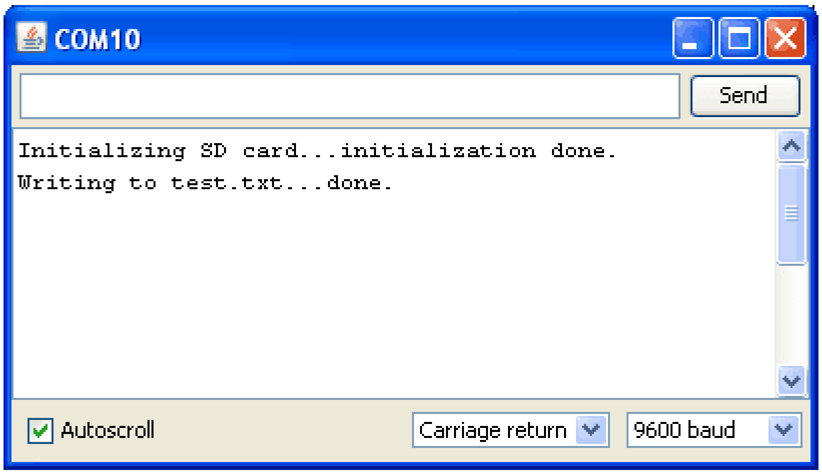
It's mostly gibberish, but it's useful to see the Volume type is FAT16 part as well as the Size of the card (about 2 GB which is what it should be) etc. .

Reading Files

The following sketch will do a basic demonstration of writing to a file. This is a common desire for data logging and such.

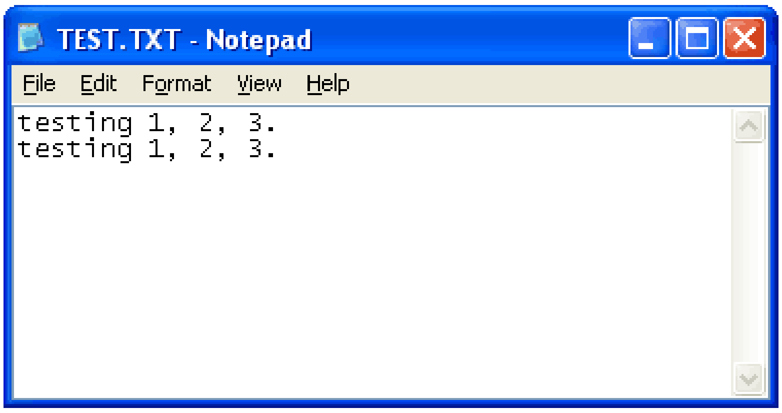
```
#include <SD.h>

File myFile;

void setup()
{
  Serial.begin(9600);
  Serial.print("Initializing SD card...");
  // On the Ethernet Shield, CS is pin 4. It's set as an output by default.
  // Note that even if it's not used as the CS pin, the hardware SS pin
  // (10 on most Arduino boards, 53 on the Mega) must be left as an output
  // or the SD library functions will not work.
   pinMode(10, OUTPUT);

  if (!SD.begin(10)) {
    Serial.println("initialization failed!");
    return;
  }
  Serial.println("initialization done.");

  // open the file. note that only one file can be open at a time,
  // so we have to close this one before opening another.
  myFile = SD.open("test.txt", FILE_WRITE);

  // if the file opened okay, write to it:
  if (myFile) {
    Serial.print("Writing to test.txt...");
    myFile.println("testing 1, 2, 3.");
	// close the file:
    myFile.close();
    Serial.println("done.");
  } else {
    // if the file didn't open, print an error:
    Serial.println("error opening test.txt");
  }
}

void loop()
{
	// nothing happens after setup
}
```

When we run it we should see the following:



```
Initializing SD card...initialization done.
Writing to test.txt...done.
```

We can then open up the file in operating system by inserting the card. We'll see one line for each time the sketch ran. That is to say, it **appends** to the file, not overwriting it.



```
testing 1, 2, 3.
testing 1, 2, 3.
```

Some things to note:

- We can have multiple files open at a time, and write to each one as we wish.
- We can use **print** and **println()** just like Serial objects, to write strings, variables, etc
- We must **close ()** the file(s) when we're done to make sure all the data is written permanently!

- We can open files in a directory. For example, if we want to open a file in the directory such as **/MyFiles/example.txt** we can call **SD.open("/myfiles/example.txt")** and it will do the right thing

Reading from Files

Next up we will show how to read from a file, it's very similar to writing in that we **SD.open()** the file but this time we don't pass in the argument **FILE_WRITE** this will keep we from accidentally writing to it. We can then call **available ()** (which will let we know if there is data left to be read) and **read ()** from the file, which will return the next byte.

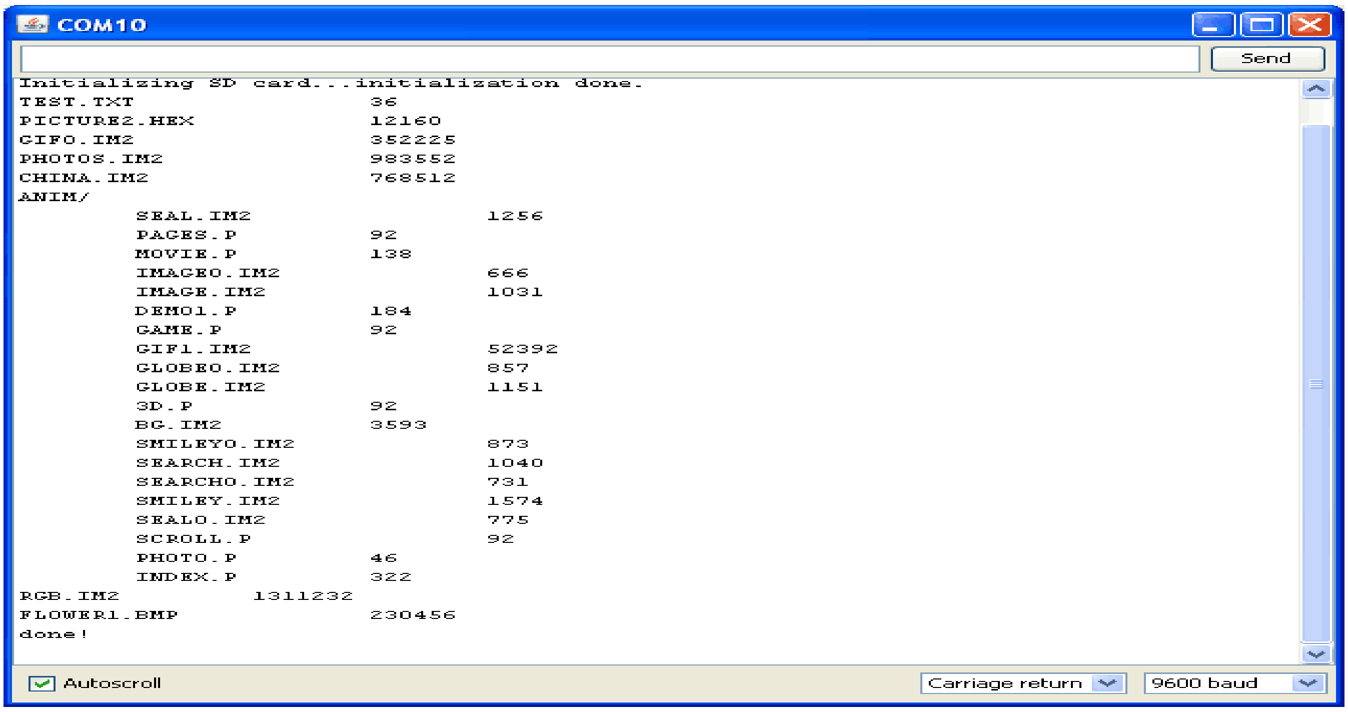
```
#include <SD.h>

File myFile;

void setup()
{
  Serial.begin(9600);
  Serial.print("Initializing SD card...");
  // On the Ethernet Shield, CS is pin 4. It's set as an output by default.
  // Note that even if it's not used as the CS pin, the hardware SS pin
  // (10 on most Arduino boards, 53 on the Mega) must be left as an output
  // or the SD library functions will not work.
   pinMode(10, OUTPUT);

  if (!SD.begin(10)) {
    Serial.println("initialization failed!");
    return;
  }
  Serial.println("initialization done.");

  // open the file for reading:
  myFile = SD.open("test.txt");
  if (myFile) {
    Serial.println("test.txt:");

    // read from the file until there's nothing else in it:
    while (myFile.available()) {
           Serial.write(myFile.read());
    }
    // close the file:
    myFile.close();
  } else {
        // if the file didn't open, print an error:
    Serial.println("error opening test.txt");
  }
}

void loop()  {  }Some things to note:
```

- We can have multiple files open at a time, and read from each one as we wish.
- **Read()** only returns a byte at a time. It does not read a full line or a number!
- We should **close()** the file(s) when we're done to reduce the amount of RAM used.

Recursively Listing/Reading Files

In the latest version of the SD library, we can *recurse* through a directory and call **openNextFile**() to get the next available file. These aren't in alphabetical order, they're in order of creation .

- To see it, run the **SD→listfiles** example sketch
- Here we can see that we have a subdirectory **ANIM** (we have animation files in it). The numbers after each file name are the size in bytes of the file.



```
Initializing SD card...initialization done.
TEST.TXT                36
PICTURE2.HEX            12160
GIFO.IM2                352225
PHOTOS.IM2              983552
CHINA.IM2               768512
ANIM/
        SEAL.IM2                1256
        PAGES.P         92
        MOVIE.P         138
        IMAGEO.IM2              666
        IMAGE.IM2               1031
        DEMO1.P         184
        GAME.P          92
        GIF1.IM2                52392
        GLOBEO.IM2              857
        GLOBE.IM2               1151
        3D.P            92
        BG.IM2          3593
        SMILEYO.IM2             873
        SEARCH.IM2              1040
        SEARCHO.IM2             731
        SMILEY.IM2              1574
        SEALO.IM2               775
        SCROLL.P                92
        PHOTO.P         46
        INDEX.P         322
RGB.IM2         1311232
FLOWER1.BMP             230456
done!
```

Other Useful Functions

There's a few useful things we can do with SD objects we'll list a few here:

- If we just want to check if a file exists, use SD.exists("filename.txt") which will return true or false.
- We can delete a file by calling SD.remove("unwanted.txt") - be careful! This will really delete it, and there's no 'trash can' to pull it out of.
- We can create a subdirectory by calling SD.mkdir("/mynewdir") handy when we want to stuff files in a location. Nothing happens if it already exists but we can always call SD.exists() above first.

Also, there's a few useful things we can do with File objects:

- We can seek() on a file. This will move the reading/writing pointer to a new location.

  For example seek(0) will take we to the beginning of the file, which can be very

  handy! Likewise we can call position() which will tell we where we are in the file.

- If we want to know the size of a file, call size() to get the number of bytes in the file.
- Directories/folders are special files, we can determine if a file is a directory by calling
  isDirectory().
- Once we have a directory, we can start going through all the files in the directory by
  Calling openNextFile().

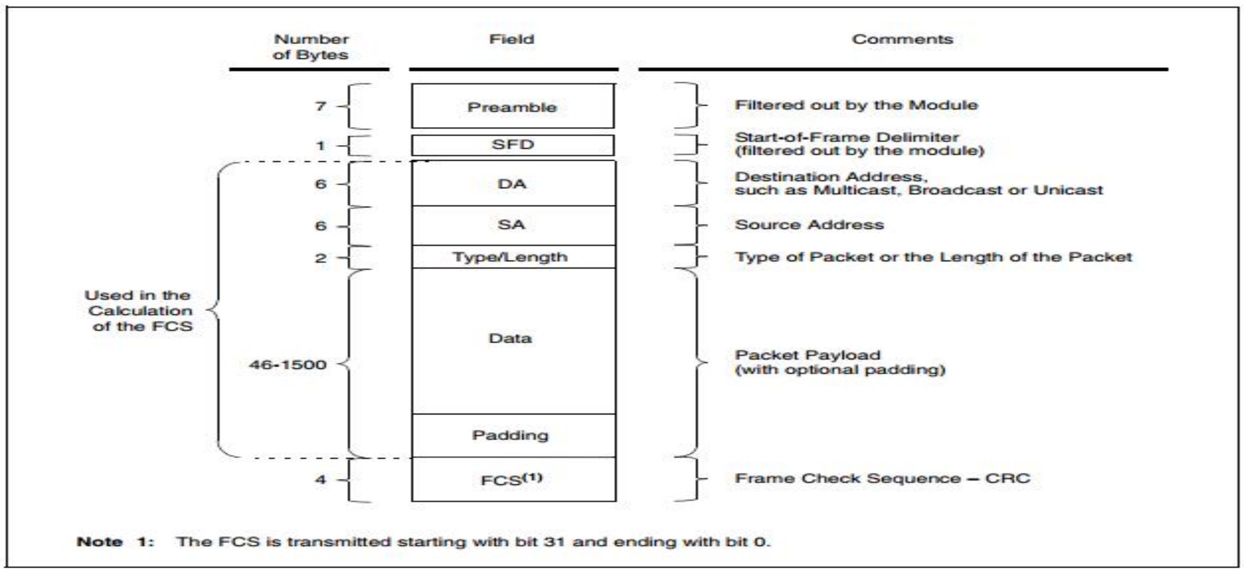
Chapter: 7

Class Discussion Topic

- Ethernet overview
- Arduino SPI interface
- Communication on Local area Network (LAN) using Arduino

Ethernet Overview

Before discussing the use of the ENC28J60 as an Ethernet interface, it may be helpful to review the structure of a typical data frame. IEEE Standard 802.3 is the basis for the Ethernet protocol 5.1. The Packet Format for Normal IEEE 802.3 compliant Ethernet frames are between 64 and 1518 bytes long. They are made up of five or six different fields: a destination MAC address, a source MAC address, a type/length field, data payload, an optional padding field and a Cyclic Redundancy Check (CRC). Additionally, when transmitted on the Ethernet medium, a 7-byte preamble field and start-of frame delimiter byte are appended to the beginning of the Ethernet packet. Thus, traffic seen on the twisted pair cabling will appear as shown in Figure.

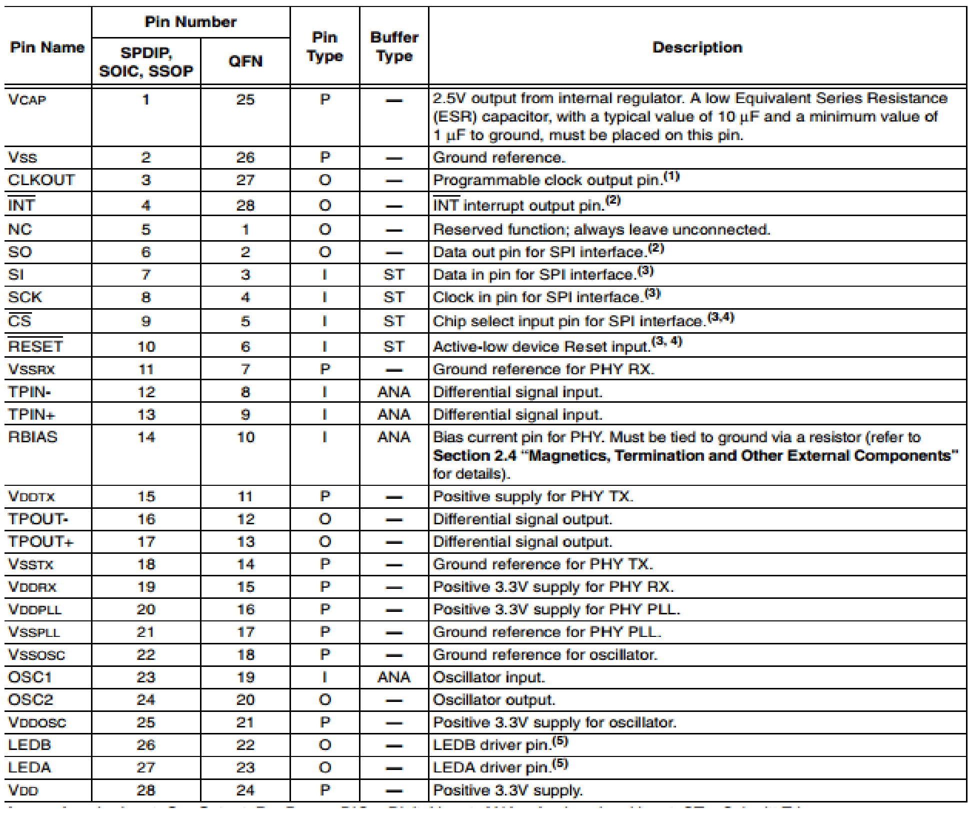

Number of Bytes
Field
Comments
7
Preamble
Filtered out by the Module
1
SFD
Start-of-Frame Delimiter (filtered out by the module)
6
DA
Destination Address, such as Multicast, Broadcast or Unicast
6
SA
Source Address
2
Type/Length
Type of Packet or the Length of the Packet
Used in the Calculation of the FCS
46-1500
Data
Padding
Packet Payload (with optional padding)
4
FCS(1)
Frame Check Sequence – CRC
Note 1: The FCS is transmitted starting with bit 31 and ending with bit 0.

ENC28J60 Ethernet Module

The ENC28J60 is a stand-alone Ethernet controller with an industry standard Serial Peripheral Interface (SPI). It is designed to serve as an Ethernet network interface for any controller equipped with SPI. The ENC28J60 meets all of the IEEE 802.3 specifications. It incorporates a number of packet filtering schemes to limit incoming packets. It also provides an internal DMA module for fast data throughput and hardware assisted checksum calculation, which is used in various network protocols. Communication with the host controller is implemented via an interrupt pin and the SPI, with clock rates of up to 20 MHz.

ENC28J60 Pin-out I/O Descriptions

The following chart is showing the complete details of I/O pins of ENC28J60 LAN module.



Pin Name	Pin Number		Pin Type	Buffer Type	Description
	SPDIP, SOIC, SSOP	QFN			
VCAP	1	25	P	—	2.5V output from internal regulator. A low Equivalent Series Resistance (ESR) capacitor, with a typical value of 10 µF and a minimum value of 1 µF to ground, must be placed on this pin.
VSS	2	26	P	—	Ground reference.
CLKOUT	3	27	O	—	Programmable clock output pin.(1)
$\overline{INT}$	4	28	O	—	$\overline{INT}$ interrupt output pin.(2)
NC	5	1	O	—	Reserved function; always leave unconnected.
SO	6	2	O	—	Data out pin for SPI interface.(2)
SI	7	3	I	ST	Data in pin for SPI interface.(3)
SCK	8	4	I	ST	Clock in pin for SPI interface.(3)
$\overline{CS}$	9	5	I	ST	Chip select input pin for SPI interface.(3,4)
$\overline{RESET}$	10	6	I	ST	Active-low device Reset input.(3, 4)
VSSRX	11	7	P	—	Ground reference for PHY RX.
TPIN-	12	8	I	ANA	Differential signal input.
TPIN+	13	9	I	ANA	Differential signal input.
RBIAS	14	10	I	ANA	Bias current pin for PHY. Must be tied to ground via a resistor (refer to **Section 2.4 "Magnetics, Termination and Other External Components"** for details).
VDDTX	15	11	P	—	Positive supply for PHY TX.
TPOUT-	16	12	O	—	Differential signal output.
TPOUT+	17	13	O	—	Differential signal output.
VSSTX	18	14	P	—	Ground reference for PHY TX.
VDDRX	19	15	P	—	Positive 3.3V supply for PHY RX.
VDDPLL	20	16	P	—	Positive 3.3V supply for PHY PLL.
VSSPLL	21	17	P	—	Ground reference for PHY PLL.
VSSOSC	22	18	P	—	Ground reference for oscillator.
OSC1	23	19	I	ANA	Oscillator input.
OSC2	24	20	O	—	Oscillator output.
VDDOSC	25	21	P	—	Positive 3.3V supply for oscillator.
LEDB	26	22	O	—	LEDB driver pin.(5)
LEDA	27	23	O	—	LEDA driver pin.(5)
VDD	28	24	P	—	Positive 3.3V supply.

(For more info please refer ENC2J60 datasheet)

Initialization Steps for ENC28J60

Before the ENC28J60 can be used to transmit and receive packets, certain device settings must be initialized. Depending on the application, some configuration options may need to be changed.

We need to set following registers:

- Receive Buffer
- Transmission Buffer
- Receive Filters
- Waiting For OST(Oscillator Start-up Timer)
- MAC Initialization Settings
- PHY Initialization Settings (half/full-duplex configuration)

Arduino SPI Interface

Serial Peripheral Interface (SPI) is a synchronous serial data protocol used by microcontrollers for communicating with one or more peripheral devices quickly over short distances. It can also be used for communication between two microcontrollers.

With an SPI connection there is always one master device (usually a microcontroller) which controls the peripheral devices. Typically there are three lines common to all the devices:

- MISO (Master In Slave Out) - The Slave line for sending data to the master
- MOSI (Master Out Slave In) - The Master line for sending data to the peripherals,
- SCK (Serial Clock) - The clock pulses which synchronize data transmission generated by the master

And one line specific for every device:

- SS (Slave Select) - the pin on each device that the master can use to enable and disable specific devices.

When a device's Slave Select pin is low, it communicates with the master. When it's high, it ignores the master. This allows we to have multiple SPI devices sharing the same MISO, MOSI, and CLK lines.

To write code for a new SPI device we need to note a few things:

- Is data shifted in Most Significant Bit (MSB) or Least Significant Bit (LSB) first? This is controlled by the SPI.setBitOrder() function.

- Is the data clock idle when high or low? Are samples on the rising or falling edge of clock pulses? These modes are controlled by the SPI.setDataMode() function.
- What speed is the SPI running at? This is controlled by the SPI.setClockDivider() function.

The SPI standard is loose and each device implements it a little differently. This means we have to pay special attention to the device's datasheet when writing code.

Generally speaking, there are four modes of transmission. These modes control whether data is shifted in and out on the rising or falling edge of the data clock signal (called the clock phase), and whether the clock is idle when high or low (called the clock polarity). The four modes combine polarity and phase according to this table:

Mode	Clock Polarity (CPOL)	Clock Phase (CPHA)
SPI_MODE0	0	0
SPI_MODE1	0	1
SPI_MODE2	1	0
SPI_MODE3	1	1

Once SPI parameters set correctly we just need to figure which registers are device controlled. This will be explained in the data sheet of device.

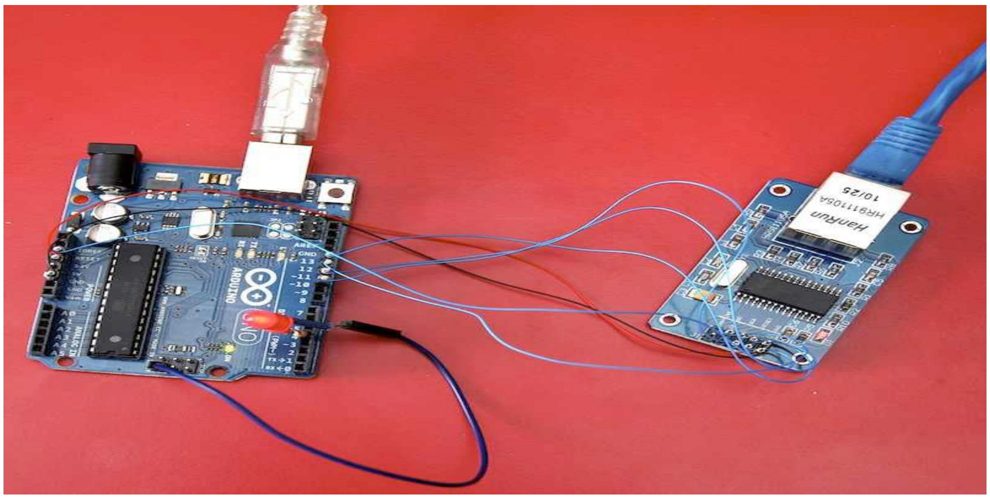
Arduino SPI Pins

The following table display on which pins the SPI lines are present on the different Arduino boards:

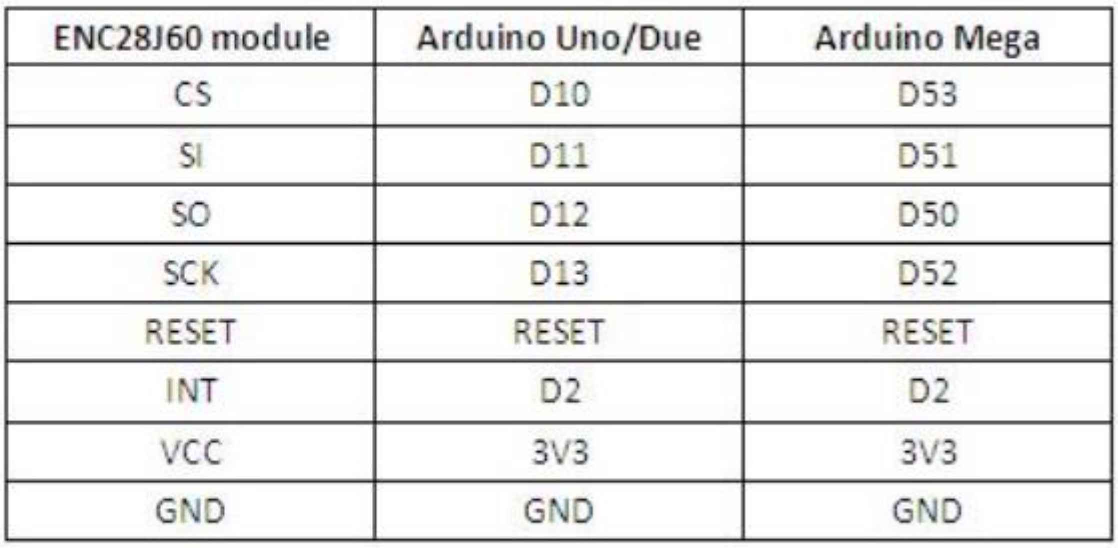
Arduino Board	MOSI	MISO	SCK	SS (slave)	SS (master)
Uno or Duemilanove	11 or ICSP-4	12 or ICSP-1	13 or ICSP-3	10	-
Mega1280 or Mega2560	51 or ICSP-4	50 or ICSP-1	52 or ICSP-3	53	-
Leonardo	ICSP-4	ICSP-1	ICSP-3	-	-
Due	ICSP-4	ICSP-1	ICSP-3	-	4, 10, 52

This Ethernet board is a simple way to give Arduino or other electronics project a network connection. It Works with all Arduino boards, including UNO, MEGA, and Nano.

Connect ENC28J60 with Arduino and Code



The following table describing which pins on the Arduino should be connected to the pins on the ENC28J60 Ethernet Module:



ENC28J60 module	Arduino Uno/Due	Arduino Mega
CS	D10	D53
SI	D11	D51
SO	D12	D50
SCK	D13	D52
RESET	RESET	RESET
INT	D2	D2
VCC	3V3	3V3
GND	GND	GND

Ethernet Library Installation

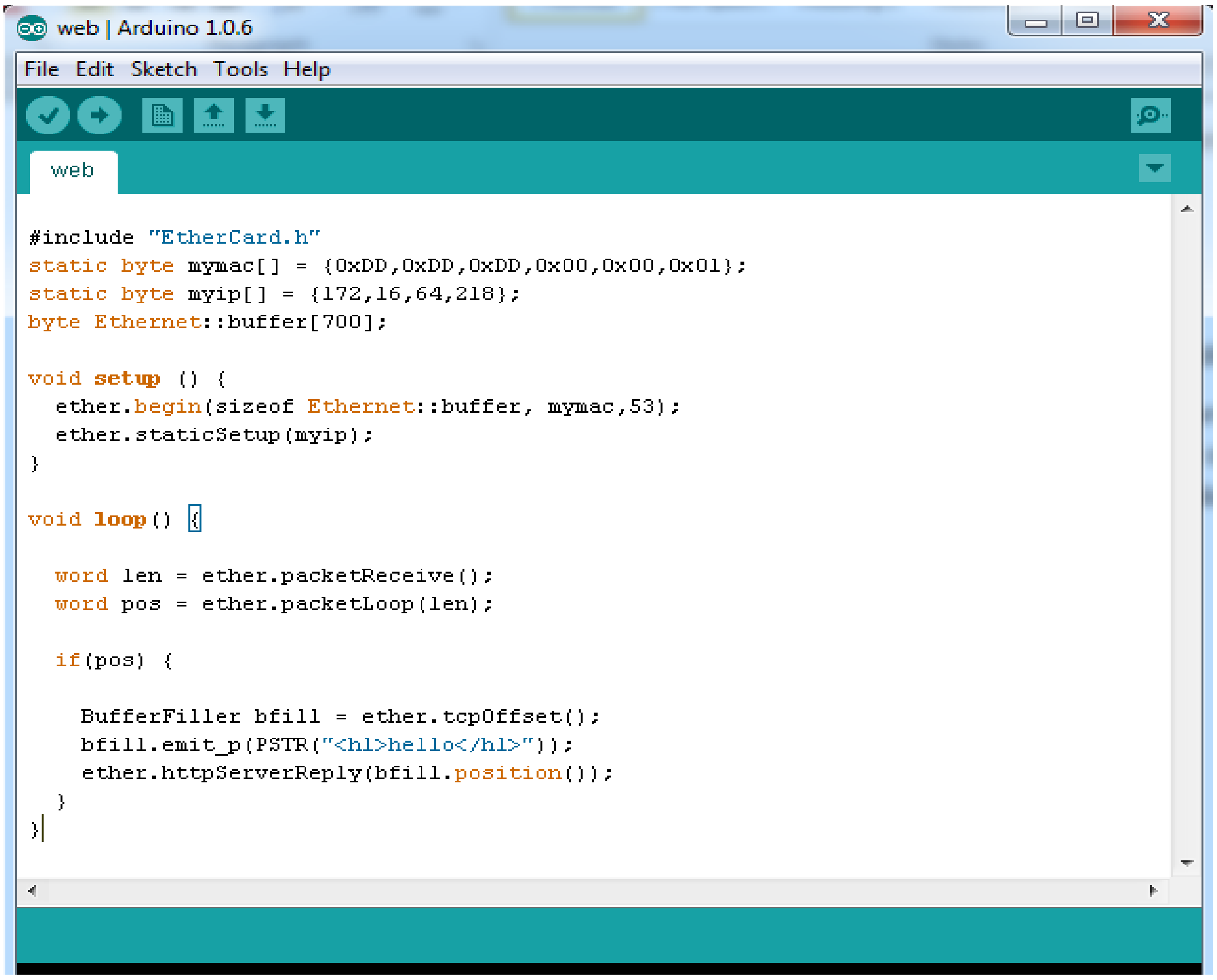
There are several libraries.

- **UIPEthernet** is for projects that require "print" to be fully implemented and need to be a drop-in replacement for the standard "Ethernet" library.
  ETHER_28j60 is great for its simplicity and small size, but comes with limitations.
  Ethercard seems best for very advanced users.
- Unzip these libraries and copy in ardiuno/libraries folder.

Example

In this example The Web browser will query inquire the Ethernet shield to return the "hello "string from Arduino board. Press F5 to refresh page.

We are using "ethercard" library to implement this example. For detailed function description please visit http://jeelabs.net/pub/docs/ethercard/

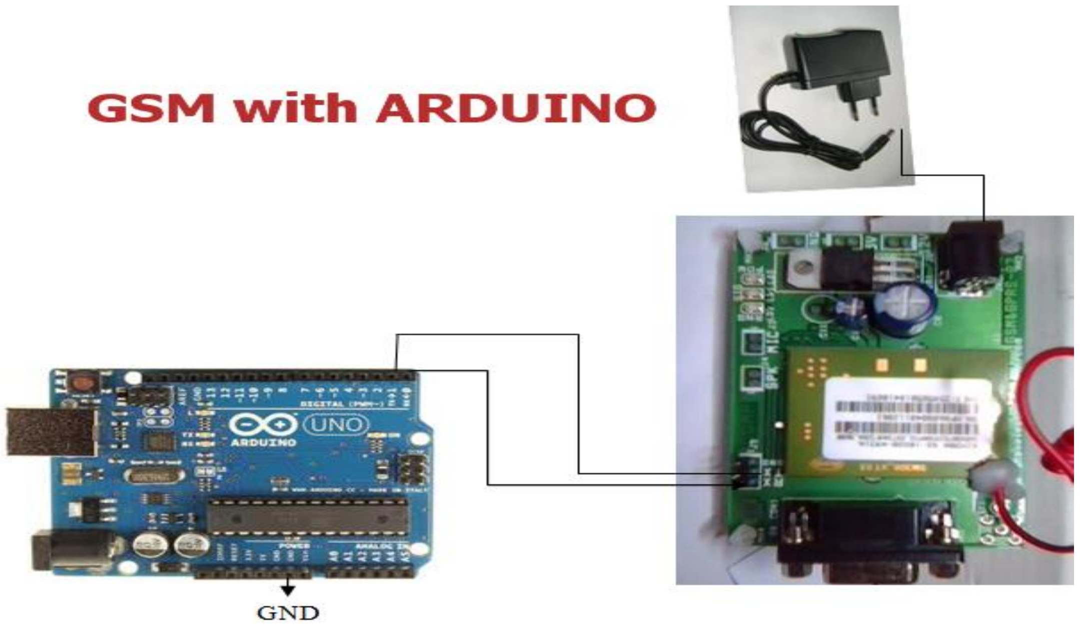


```
#include "EtherCard.h"
static byte mymac[] = {0xDD,0xDD,0xDD,0x00,0x00,0x01};
static byte myip[] = {172,16,64,218};
byte Ethernet::buffer[700];

void setup () {
  ether.begin(sizeof Ethernet::buffer, mymac,53);
  ether.staticSetup(myip);
}

void loop() {

  word len = ether.packetReceive();
  word pos = ether.packetLoop(len);

  if(pos) {

    BufferFiller bfill = ether.tcpOffset();
    bfill.emit_p(PSTR("<h1>hello</h1>"));
    ether.httpServerReply(bfill.position());
  }
}
```

Chapter: 8

Class Discussion Topic

- GSM/GPRS Communication Using Sim908
- Sms ,Dial/Receive call, HTTP/TCP Communication



1. Pin0 (RX) of Arduino connected to RX of GSM
2. Pin1 (TX) of Arduino connected to TX of GSM
3. GND of Arduino connected to GND of GSM
4. Separate power supply adapter 12v/1A used for GSM

Initialization Commands

AT	// check modem configuration
ATI	//get product name
AT+GSV	//get manufacturer info
AT+CIMI	// get device phone number
AT+GCAP	// get module capability
AT+COPS?	//Current operator

AT+CPIN?   //SIM status

AT+CSPN?   //Get the name of the service provider

AT Command Sequence for Setting up Voice Call

AT+CREG?   //Checking registration status...

ATD8586743398;   //Dialing number 8586743398

AT Command Sequence for Send Sms

AT+CMGS="7608841145"   //Send the SMS message

>enter msg text +ending character ctrl+z

AT Command Sequence for HTTP

AT+CREG?   //Check if the device is registered

AT+SAPBR=2,1   //Query if the bearer has been setupQuerying bearer 1 for getting ip

AT+HTTPINIT   //Initializing HTTP service...

AT+HTTPPARA="URL",http://www.google.com.pk "   //Setting up HTTP parameters..

AT+HTTPPARA="CID",1   //Set the CID

AT+HTTPACTION=0   //HTTP action is read

AT+HTTPREAD/   /Read the HTTP response

AT+HTTPTERM   //Terminating HTTP session..

AT Command Sequence for TCP function

```
AT+CREG?                       //Checking registration status...

AT+CGACT?                      //Checking if device is already connected...

AT+CGATT=1               //Attaching to network...

AT+CSTT="myapn"                 //Setting up APN for TCP connection...

AT+CIICR                 //Bring up GPRS Connection...

AT+CIFSR                 //Get the local IP address

AT+CIPSTART="TCP","74.124.194.252","80"

//Start TCP connection

AT+CIPSEND               //Send data.  Below data is HTTP formatted

>

GET /m2msupport/http_get_test.php HTTP/1.1

Host:www.m2msupport.net

Connection:keep-alive ctrl+z
```