

Workshop on Embedded Systems

Workshop Manual



Koshish Foundation Research Lab (KFRL)

Computer and Information Systems Engineering Department,
NED University of Engineering and Technology, Pakistan

Contents

Chapter: 01	3
Introduction to Microcontrollers & Arduino	3
□ Microcontrollers	3
□ Getting Started with Arduino	4
□ Install Arduino IDE Software	4
□ Install the Drivers for Windows	4
□ The Integrated Development Environment (IDE)	5
□ Upload a Sketch	6
Chapter: 02	8
Arduino GPIO Handling	8
□ Digital input & Output	8
□ Digital Inputs	8
□ Digital Output	9
□ Analog Input & Output	10
Chapter: 03	13
Serial Communication with Arduino UNO	13
□ Serial Communication	13
□ Serial Communication Agreements	13
□ What Do the Serial Voltage Changes Mean?	13
□ Serial Communication in Arduino	14
Chapter: 04	13
Interrupts to Arduino	16
□ External Hardware Interrupts	16
□ External Hardware Interrupts with Arduino	16
□ Special Cases	18
Chapter: 05	19
Introduction to XBee & its Interfacing with Arduino	19
□ Wireless Communication & Zigbee Basics	19
□ How to Use X-CTU	19
□ Wireless Doorbell Project	24
Chapter: 06	27
SD Card Interfacing with Arduino	27
□ Introduction	27

□ SD Card Interfacing with Arduino	27
□ Formatting SD Card.....	27
□ Wiring.....	27
□ Arduino Library & First Test.....	28
□ Reading Files	30
□ Recursively Listing/Reading Files.....	32
□ Other Useful Functions.....	33
Chapter: 07.....	34
Interfacing Arduino with Ethernet Shield.....	34
□ Ethernet Overview	34
□ ENC28J60 Ethernet Module.....	34
□ ENC28J60Pin-out I/O Description.....	35
□ Initialization Steps for ENC28J60.....	36
□ Arduino SPI Interface	36
□ Arduino SPI Pins	37
□ Connect ENC28J60 with Arduino and Code.....	38
□ Ethernet Library Installation.....	38
Chapter: 08.....	40
Exploring GPRS Module.....	40
□ Introduction	40
□ Initialization Commands.....	40
□ AT Command Sequence for Setting up Voice Call.....	41
□ AT Command Sequence for Send SMS	41
□ AT Command Sequence for HTTP	41
□ AT Command Sequence for TCP function.....	42

Chapter: 01

Introduction to Microcontrollers & Arduino

➡ Microcontrollers

The simple difference between microprocessor and microcontroller 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 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 general requirements of a PC, i.e. performing 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 notes mentioned as follow,

Microprocessor = Central Processing Unit (CPU)

Micro controller = CPU+ Peripherals + Memory

Peripherals = Ports + Clock + Timers + Analog to Digital converters +LCD drivers + Digital to Analog converter + Other stuff

Memory = EEPROM + EPROM + Flash

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



Fig. 01: Microcontroller ICs

➡ Getting Started with Arduino

Arduino Uno board will be used throughout this course. The Arduino Uno is a microcontroller board based on the ATmega328 Microcontroller.

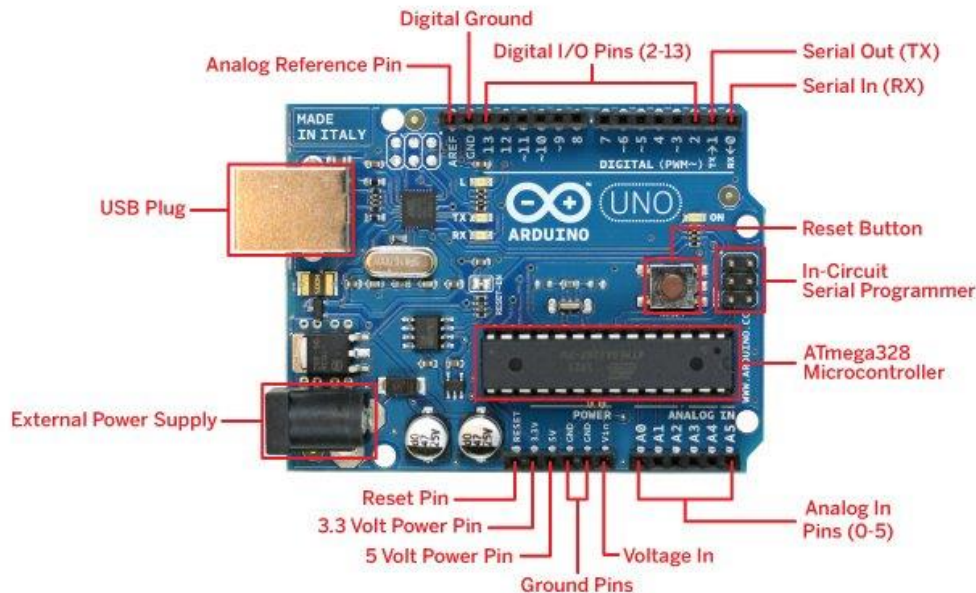


Fig. 02 Arduino UNO board with Atmega328 microcontroller

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 Arduino IDE Software

The software and step-by-step directions for its installation is 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 i.e., 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). There must be 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.
- 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.

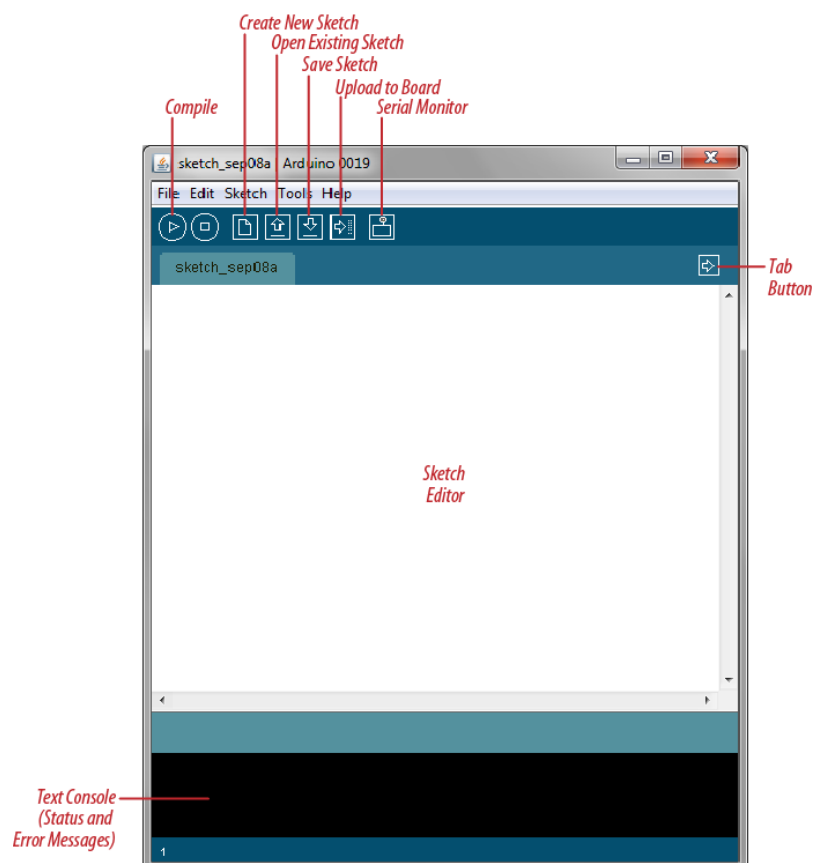


Fig. 03 Arduino integrated development environment (IDE)

- ✓ **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 transmits it over the USB cable to board.
- ✓ **Serial Monitor** – This is used to receive data from external devices through USB cable /DB9 connector
- ✓ **Tab Button** - This lets to create multiple files in sketch. This is for more advanced programming that is not a part of this course.
- ✓ **Sketch Editor** - This is where the sketches are written.
- ✓ **Text Console** - This shows what the IDE is currently doing and error messages are also displayed here
- ✓ **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 consists of few example sketches which can be uploaded to Arduino board 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 not aware of exact port, disconnect the UNO and the entry that disappears is the right one.)
- 5) Click the Upload button.
- 6) After the message “Done uploading” appears, the LED on Arduino will start blinking once a second.

Each sketch of Arduino IDE must contain at least two functions.

- ✓ **setup ()** which is called once when the program starts.
- ✓ **loop ()** which is called repetitively over and over again as long as the Arduino.

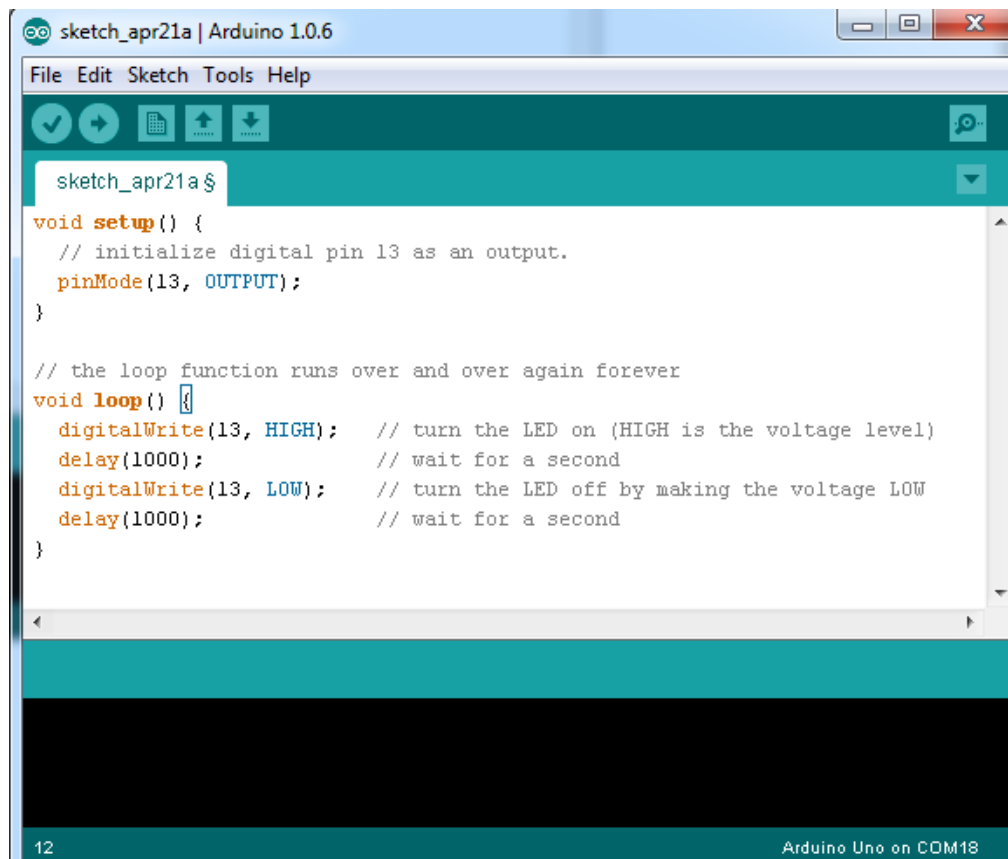


Fig. 04 Uploading Blink example to Arduino UNO board

Chapter: 02

Arduino GPIO Handling

➡ 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. 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.

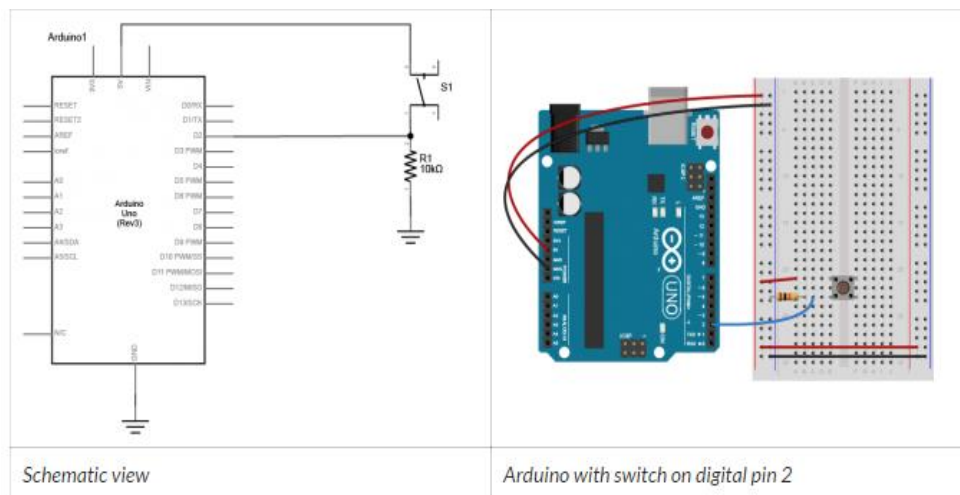


Fig. 05 External digital inputs to microcontroller

Fig. 05 shows the electrical schematic for a digital input to a microcontroller. The current has two directions it can go to ground, through resistor or microcontroller. When switch is closed, the current will follow the path of least resistance, to the microcontroller pin, and the microcontroller can then read voltage. 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(13, HIGH);
  } else {
    // if pin 2 is LOW, set pin 13 LOW:
    digitalWrite(13, LOW);
  }
}

```

➡ Digital Output

Just as digital inputs sense activities which have two states, digital or binary outputs allow controlling 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.

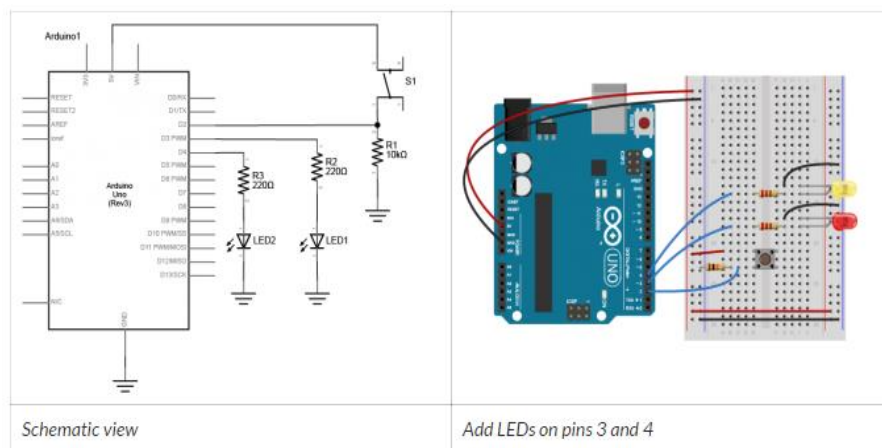


Fig. 06 External inputs and outputs to Arduino UNO

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 is used with 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 producing high-current then the 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 section, a variable resistor will be connected to a microcontroller and will serve as an analog input. Through this changing conditions from the physical world can be monitored and then converted to changing variables in a program. Many of the useful sensors are analog in nature. They deliver a variable voltage, which can be read on analog input pins using the `analogRead ()` command.

✓ Prepare the Breadboard

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

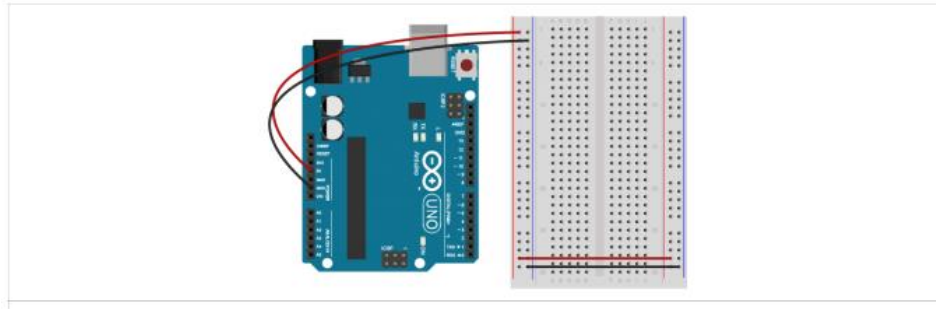


Fig. 07 Power connection with Arduino UNO

Connect a potentiometer to analog pin 0 of Arduino, LED and a resistor to digital pin 9 as shown in Fig. 08.

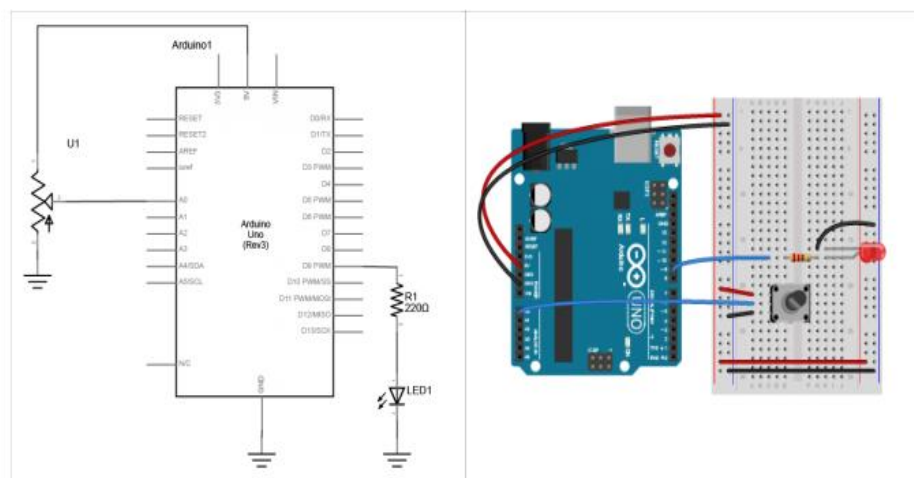


Fig. 08 Connecting Analog inputs to Arduino UNO

✓ Program the Module

First, establish some global variables; one to hold the value returned by 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 analog Read() 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 and print out the brightness value. When this code will be uploaded to microcontroller, the LED should dim up and down as the pot is turned, and the brightness value should show up in the serial monitor.

✓ Code

```
const int ledPin = 9;    // pin that the LED is attached to
int analogValue = 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: 03

Serial Communication with Arduino UNO

➡ Serial Communication

In order to make two devices communicate, whether they are desktop computers, microcontrollers, or any other form of computer, a method of communication and an agreed-upon language is needed. 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 the same rate. This is known as asynchronous serial communication. There is not any 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. For example, let's say two devices are to exchange data at a rate of 9600 bits per second.

- 1) Make sure there is an agreed upon high and low voltage supplying each device.
- 2) Make three connections between the two devices: a common ground connection, so that both devices have a common reference point to measure voltage by.
- 3) One wire for the sender to send data to the receiver on and one wire for the receiver to send data to the sender on.

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. 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 it is required to send the number 90 from one device to another. Then firstly, the number needs to be converted from decimal representation 90 to a binary representation which is equivalent to 01011010.

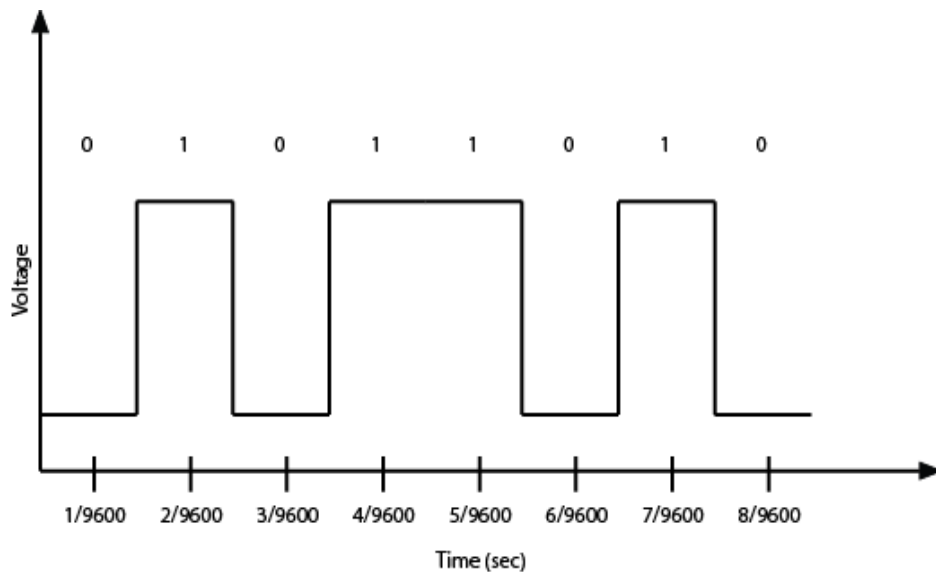


Fig. 09 Serial transmission of binary data

As clear from diagram given in Fig. 09, two devices have to agree on the order of 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 there is an agreed upon voltage, data rate, order of interpretation of bits, and agreement on what the voltage levels mean, any data can be exchanged serially.

For data transmission as aforementioned, 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 logic 0, and a low voltage indicates logic 1. It is 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

This type of communication is used between an Arduino and a workstation. It takes place via the Arduino board's serial or USB connection and on digital pins 0 (RX) and 1 (TX) but at one time only one way of serial communication can be adopted. Therefore, while using following mentioned serial functions, one cannot use Rx/Tx pins of the microcontroller.

- Serial.begin(speed)
- intSerial.available()
- intSerial.read()

- Serial.flush()
- Serial.print(data)
- Serial.println(data)

✓ Example Program

In this example a potentiometer is interfaced with analog pin A1 for input and digital pin2 is considered as output. Serially transmit analog data and receive it back by connecting TX and RX i.e., loop back condition.

✓ 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();
    } }
    voidestablishContact() {
      while (Serial.available() <= 0) {
        Serial.print(255); // send a 255
        delay(300);
      }
    }
  }
}
```


Chapter: 04

Interrupts to 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 below mentioned tasks:

- 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 that 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. In a standard Arduino board, these are tied to digital pins 2 and 3, respectively. Let's enable INT0 so that it can detect an input change on pin 2 from a button or switch.

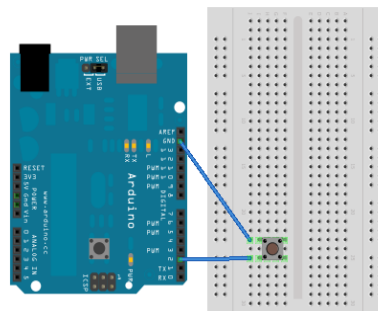


Fig. 10 Connecting switch to Arduino interrupt pin

✓ 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;
}
```

Afore given is a sample program for attaching an interrupt to Arduino input pin. The label "LED" to the compiler is defined as the number 13. So digitalWrite (LED, HIGH) is the same as digitalWrite (13, HIGH) which is the same as digitalWrite(LED, 1).

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 i.e., bit 0.

In setup (), the function **attachInterrupt(interrupt, function, mode)** is used to activate the pin as an input interrupt. Where "interrupt" shows the interrupt number 0 to 5; "function" is known as the interrupt service routine (ISR) i.e., 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.

Therefore in the sample code, attachInterrupt (0, toggle, FALLING) zero corresponds to interrupt 0 on pin 2, toggle() is the ISR routine at the bottom of the program, and RISING means when an electrical signal goes from 0V to 5V then toggle() ISR performs its function. What started as state = LOW is now state = HIGH and vice-versa. The loop() will simply keep writing the variable "state" to the LED on pin 13, 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 LED1 is off press SW0 and LED will come on, release nothing happens. Press again and the LED is off. With `attachInterrupt(interrupt, function, mode)` there are four "mode" declarations can be defines which are explained as under:

- 1) **RISING** to trigger when the pin goes from low to high.
- 2) **FALLING** for when the pin goes from high to low.
- 3) **CHANGE** to trigger the interrupt whenever the pin changes value
- 4) **LOW** to trigger the interrupt whenever the pin is low.

➡ **Special Cases**

It is possible to reassign interrupts using the **`attachInterrupt()`** function again, but it is also possible to remove them by calling the function **`detachInterrupt()`**. If there is a section of code that is time sensitive, and it is important that an interrupt must not be called during that time frame. Then one 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: 05

Introduction to XBee & its Interfacing with Arduino

➡ Wireless Communication & Zigbee Basics

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

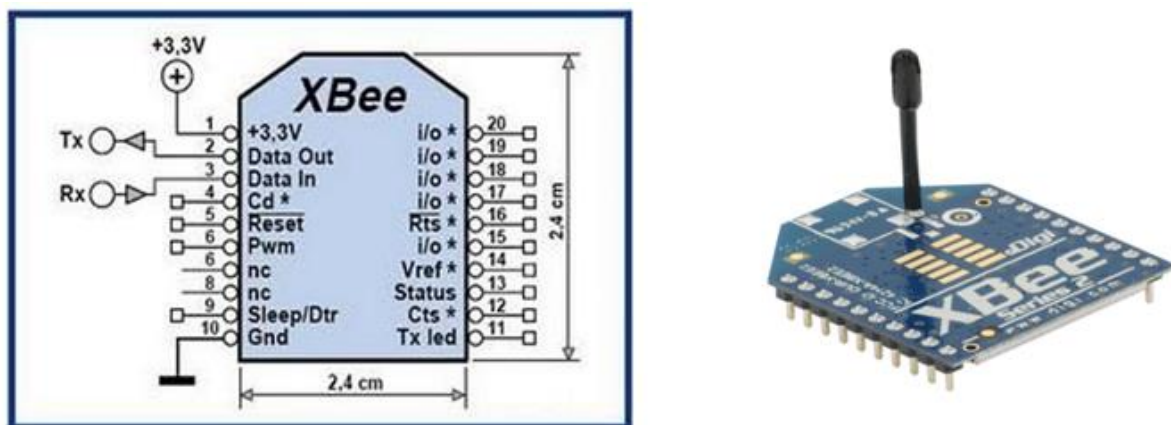


Fig. 11 XBee module and its pin configuration

➡ How to Use X-CTU

XCTU is free software created by Digi International to aid in configuring XBee modules, either individually or as a network (refer to Fig. 12). It is extremely powerful and versatile. Beginners are recommended to use this software to familiarize themselves with the working of XBee before attempting to use AT Commands. Once the XBee is connected to the computer using a USB port, select one of the options in the upper left corner namely “Add Devices” and “Discover Devices”. The “Add Devices” option searches for a device connected to a single port with the given settings (refer to Fig. 12) whereas the “Discover Devices” can be used to search multiple ports with different combinations of settings to retrieve a list of all devices connected with the above setting (refer to Fig. 13).

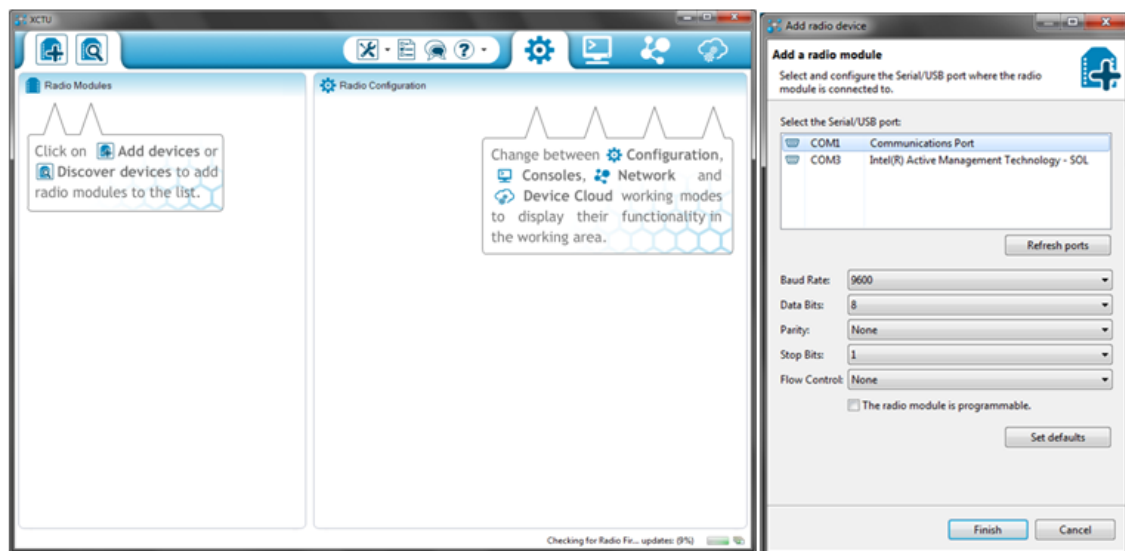


Fig. 12 XCTU environment and Add radio devices

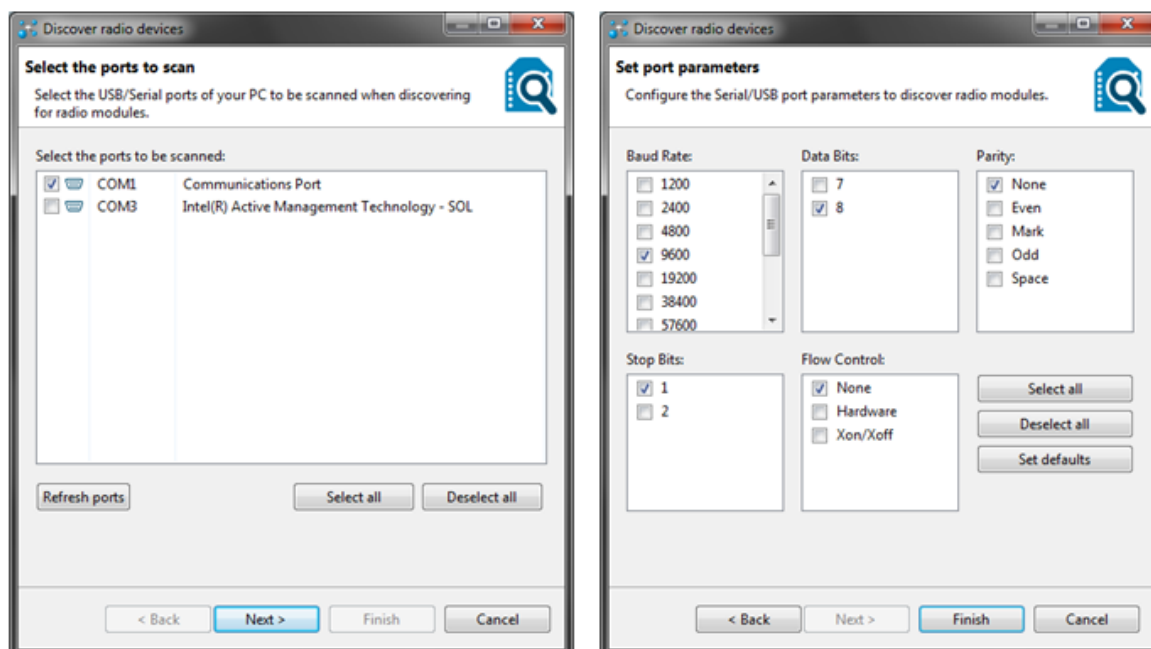


Fig. 13 Discovering radio modules

Once the “Finish” button is pressed, XCTU searches the selected ports for an XBee device and displays it in the left column. Clicking it will start a read operation and all the firmware configurations will be shown in the right column (please see Fig. 14). Now, to change one of the settings that have been read from the XBee module, simply click the editable field of that particular setting and enter a new value. To commit the change to the XBee module’s firmware, click on the pencil icon next to the field to write it or use the pencil icon on the top to commit all changes to the firmware.

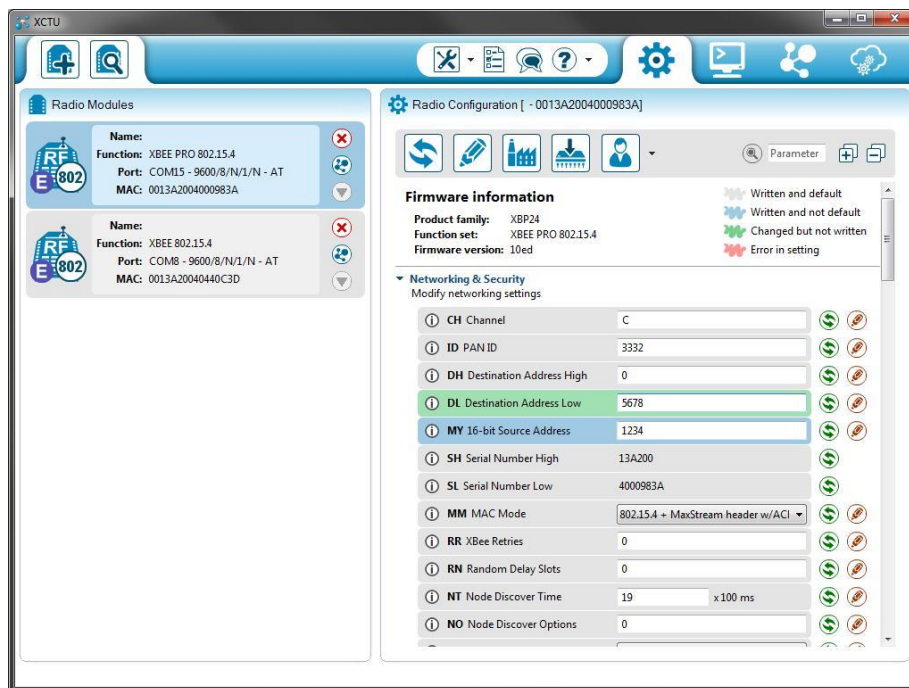


Fig. 14 Parameters of connected XBee module

Firmware can be uploaded by using the Update Firmware button on top of the Radio Configuration pane (refer to Fig. 15).



Fig. 15 Update firmware to connected XBee module

A window will pop up with a list of all the radio modules and related firmware that can be uploaded. Upload the required firmware by selecting particular product family of the connected radio module.

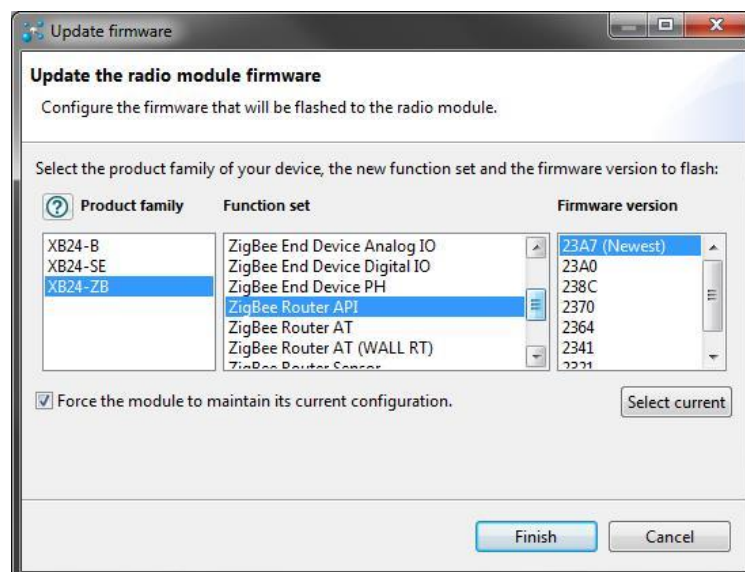


Fig. 16 Uploading required firmware according to particular product family

In background XCTU uses AT commands to configure the XBee firmware. These are simple strings that are sent using UART to the XBee's DIN (Rx) pin. These are helpful when XBees are to be configured by a microcontroller or computer application. AT Commands to work, the XBee module must be in AT Command Mode. To enter this mode transmit "+++" (without the quotes) serially to radio module. A carriage return should not be sent after the plus signs. Once "OK" is received, the module has entered AT Command Mode and can be configured.

The general AT command format is shown in Fig. 17. After every command, the module will return an "OK" if the command has been executed successfully.

"AT" Prefix	ASCII Command (ex. Name of the field)	Space (Optional)	Parameter/Value (Optional, Hex)	"\n" Carriage Return
----------------	--	---------------------	------------------------------------	----------------------------

Fig. 17 AT Command Structure

Few AT commands are listed as under:

- ✓ ATMY: Sets the 16 bit address of the module
- ✓ ATDH: Sets the DH register
- ✓ ATDL: Sets the DL register
- ✓ ATCN: Exits AT Command mode

Open the COM port using a program like Putty, HyperTerminal or XCTU and type the following commands to configure the modules.

✓ **XBee Destination**

- +++
- ATMY 5678
- ATDH 0
- ATDL 1234
- ATCN

✓ **XBee Source**

- +++
- ATMY 1234
- ATDH 0
- ATDL 5678
- ATCN

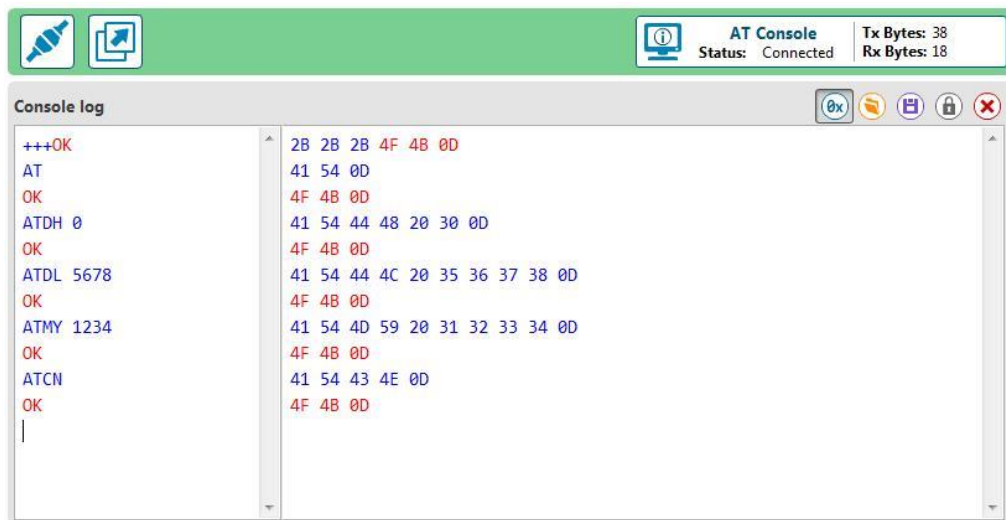


Fig. 18 Configuring XBee using AT commands

✓ Transfer of Data

Data transfer between the modules can be accomplished using the XCTU software. Connect both of the modules to the computer. Select one of the modules and click on the terminal tab in the right pane. Click on the Open Connection button (plug icon) in the top left to open the COM port. This will establish a connection between XCTU and the module. The column on the left is used to enter data to be sent to the connected module's DIN (Rx) pin. A hexadecimal representation of it will be shown in the right column. Blue represents data sent to the module and red represent data received from the module.

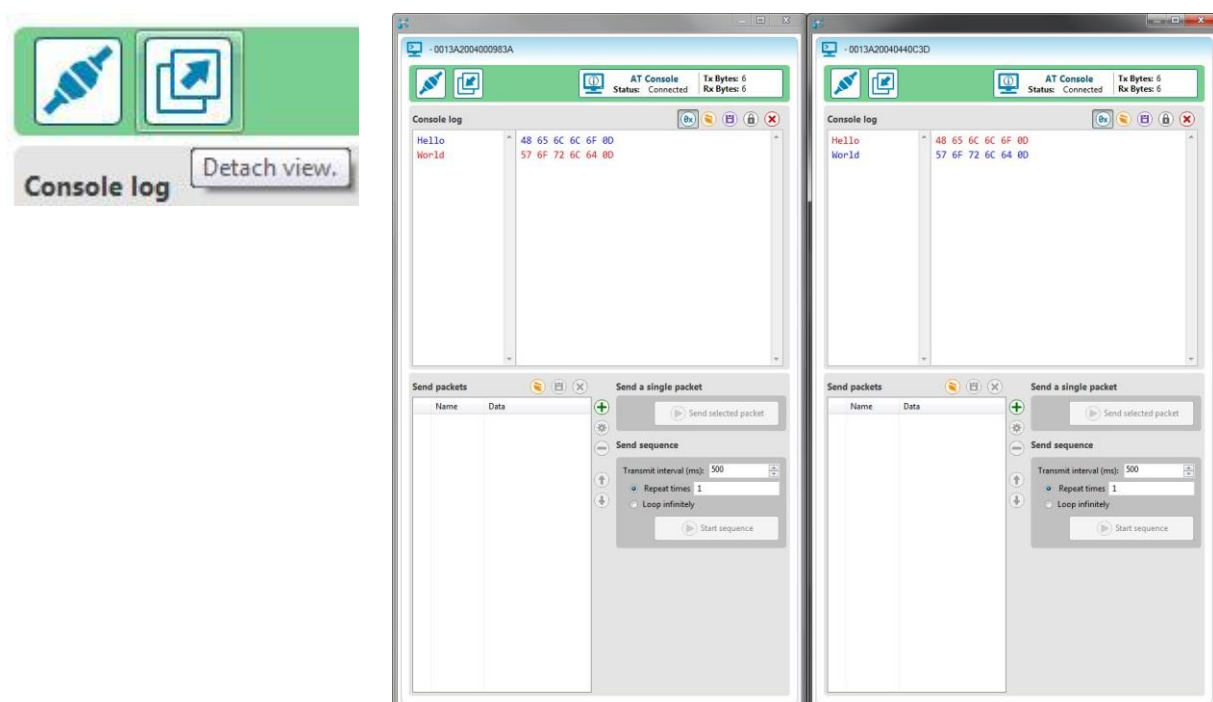


Fig. 19 Transferring data from connected radio modules

The tabs in the terminal view may be popped out when monitoring multiple modules using the Detach View button next to the Open Connection button.

➡ Wireless Doorbell Project

A small project on wire-less doorbell system is explained in this section. Its main components are Arduino boards, XBee modules, switch, and buzzer as shown in Fig. 20. When the switch is pressed the data will be transmitted to remote Arduino and a buzzer connected to it will sound.

✓ XBee Interfacing with Arduino

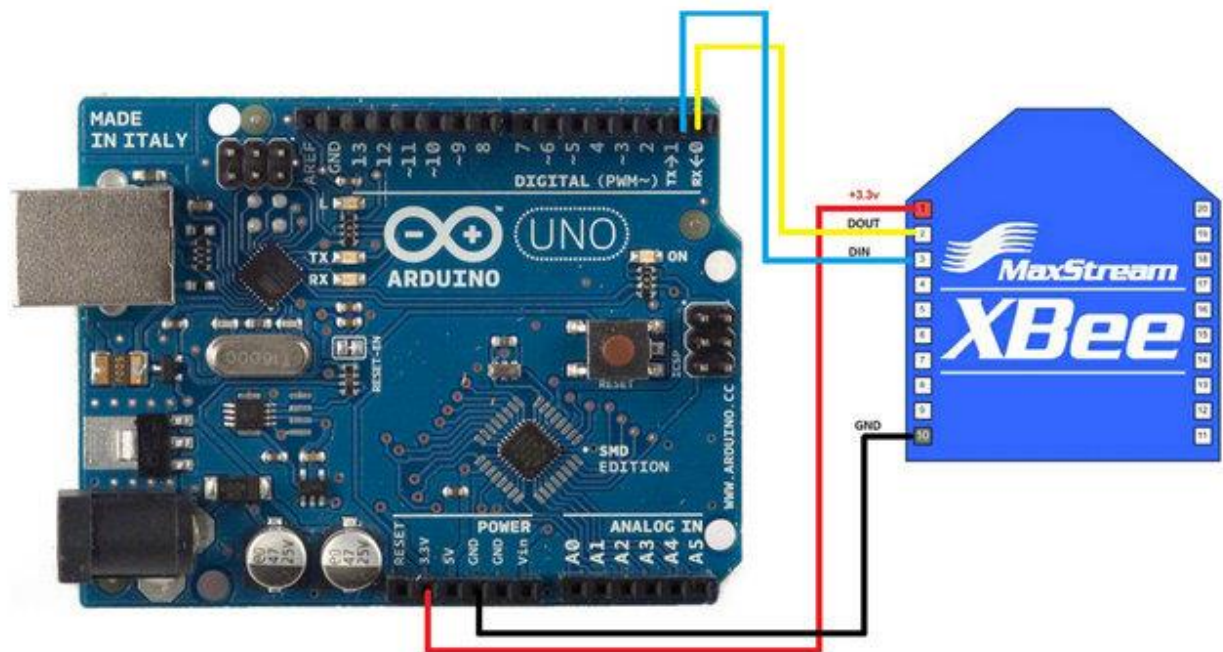


Fig. 20 Arduino interfacing with XBee radio

✓ Required Components

Following components are needed for this project:

- Hookup wires.
- Two Arduino boards.
- USB cable for the Arduino.
- 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/END node AT
- USB cable for the XBee breakout board. Every XBEE network has only one coordinator. Other nodes can be configured as routers or end nodes. It is strongly suggested that mark down the XBees to distinguish the coordinator from the router(s)/end nodes.

✓ Wireless Doorbell Connections Lay-out

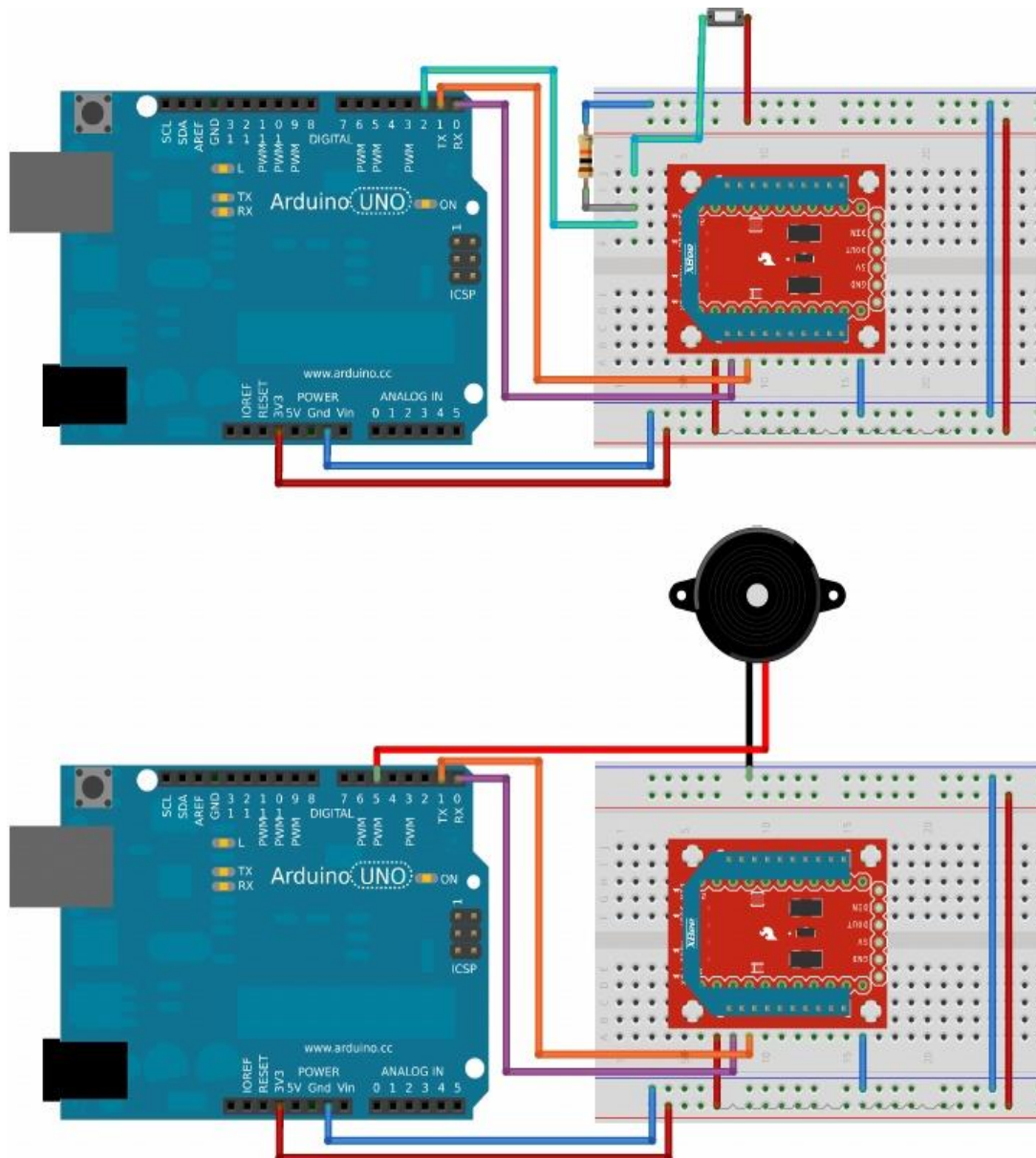


Fig. 21 Connection between different components

✓ The Button Arduino

The following code corresponds with the button side of the project. When uploading programs to Arduino, disconnect the digital pin 0 (RX) connected to XBee and then reconnect it after the loading is completed. Otherwise an error will occur. This is the transmitter side of the project.

```
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 through XBee module when the button is pressed and will ring the buzzer.

```
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
  if (Serial.available() > 0) {
    if (Serial.read() == 'D'){
      //ring the bell briefly
      digitalWrite(BELL, HIGH);
      delay(10);
      digitalWrite(BELL, LOW);}
  }
}
```

Chapter: 06

SD Card Interfacing with Arduino

➡ Introduction

When any sort of data logging, graphics or audio is carried out, at least a megabyte of storage is needed i.e., 64 Mega Bytes is probably the minimum. To get that kind of storage some external components can be used i.e., flash cards, often called as SD or microSD cards. They can pack gigabytes into a space smaller than a coin.

➡ SD Card Interfacing with Arduino

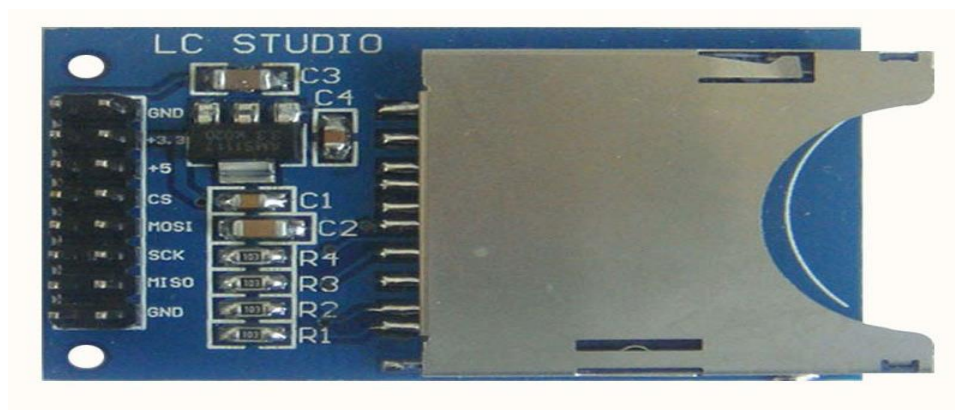


Fig. 22 SD Card reader

➡ Formatting SD Card

Even though raw SD cards can be used, it is most convenient to format the card to a file system. For the Arduino and nearly for all other SD card libraries, the card must be formatted FAT16 or FAT32. Some only allow one or the other. The Arduino SD library can use either.

➡ Wiring

Now that card is ready to use, the microSD can be connected with breakout board. It uses Arduino SPI interface. For other Arduino board such as the Duemilanove/Diecimila/Uno the pins for SPI interface are digital 13 (SCK), 12 (MISO) and 11 (MOSI). A fourth pin will also be needed for 'chip/slave select' (SS) line. Traditionally this is pin 10 but in actual any pin can be used in its place. For Mega Arduino boards, of course the pins configuration will be different.

- 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
- Connect DO to pin 12
- Connect DI to pin 11
- Connect CS to pin 10

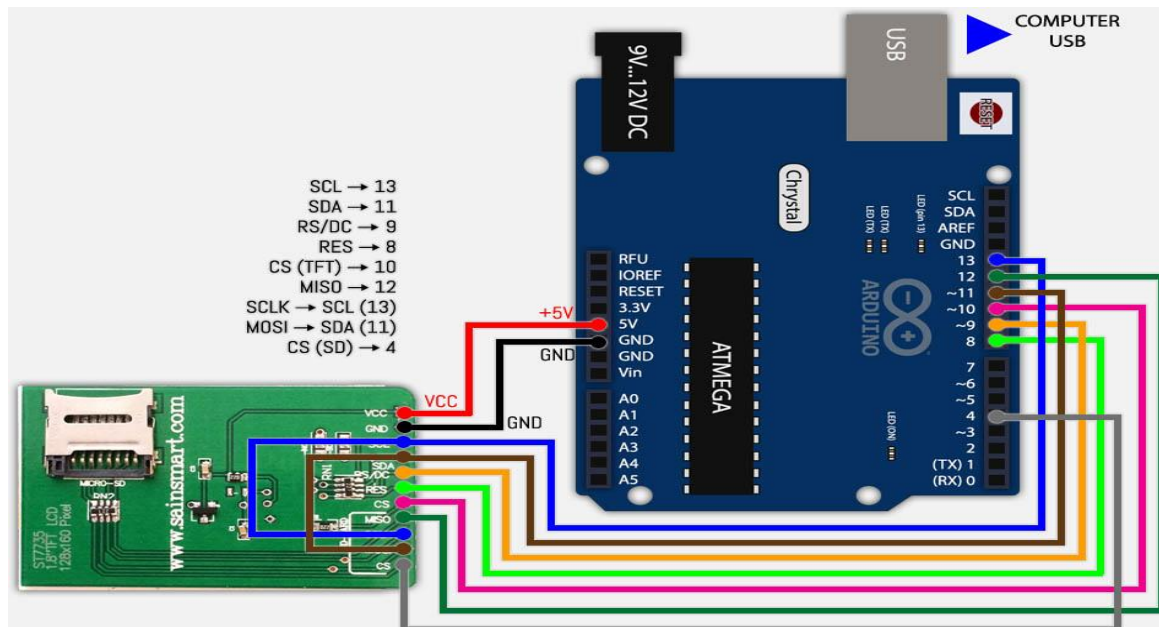


Fig. 23 Interfacing SD card reader with Arduino

➡ Arduino Library & First Test

Interfacing with an SD card is a bunch of work, but luckily there is an Adafruit customer fat16lib(William G) which has been written very nicely and it is now part of the Arduino IDE known as SD. It is available in Example submenu of Arduino IDE as shown in Fig. 24. From there select the **CardInfo**. 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 chip Select line is correct, for this wiring here we are using digital pin 10 so change it to 10. After that insert the SD card into the breakout board and upload the sketch. Open up serial monitor and type a character into the text box when prompted. You will probably get something as shown in Fig. 25.

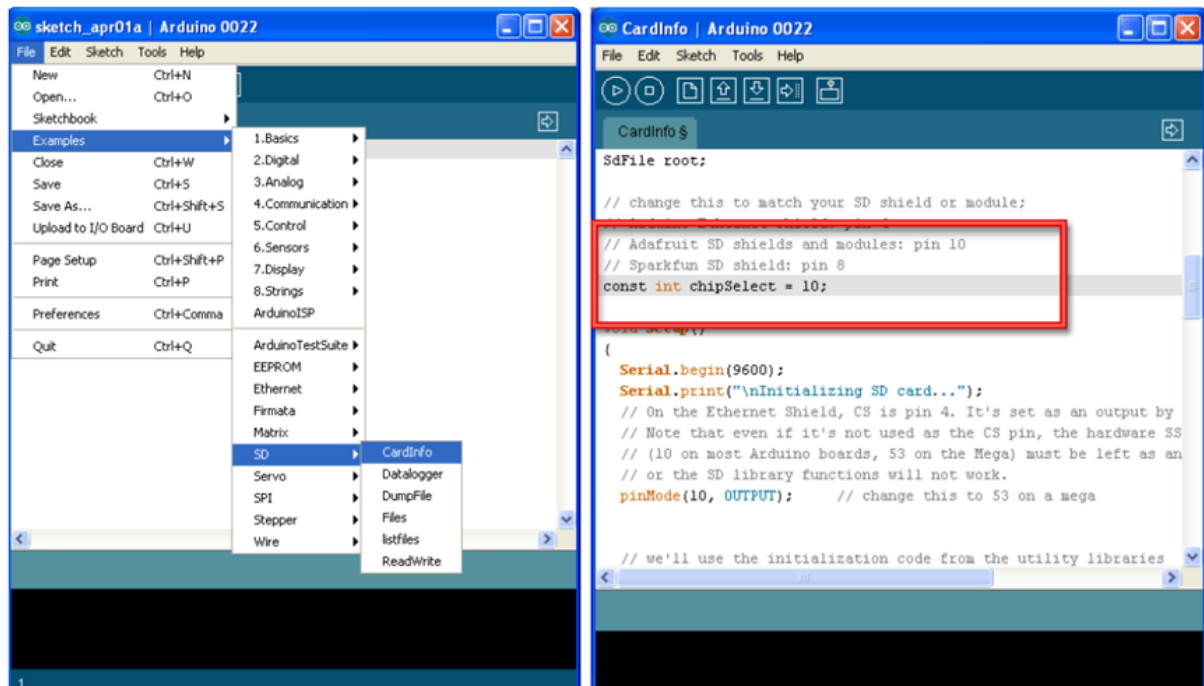


Fig. 24 SD card library for Arduino

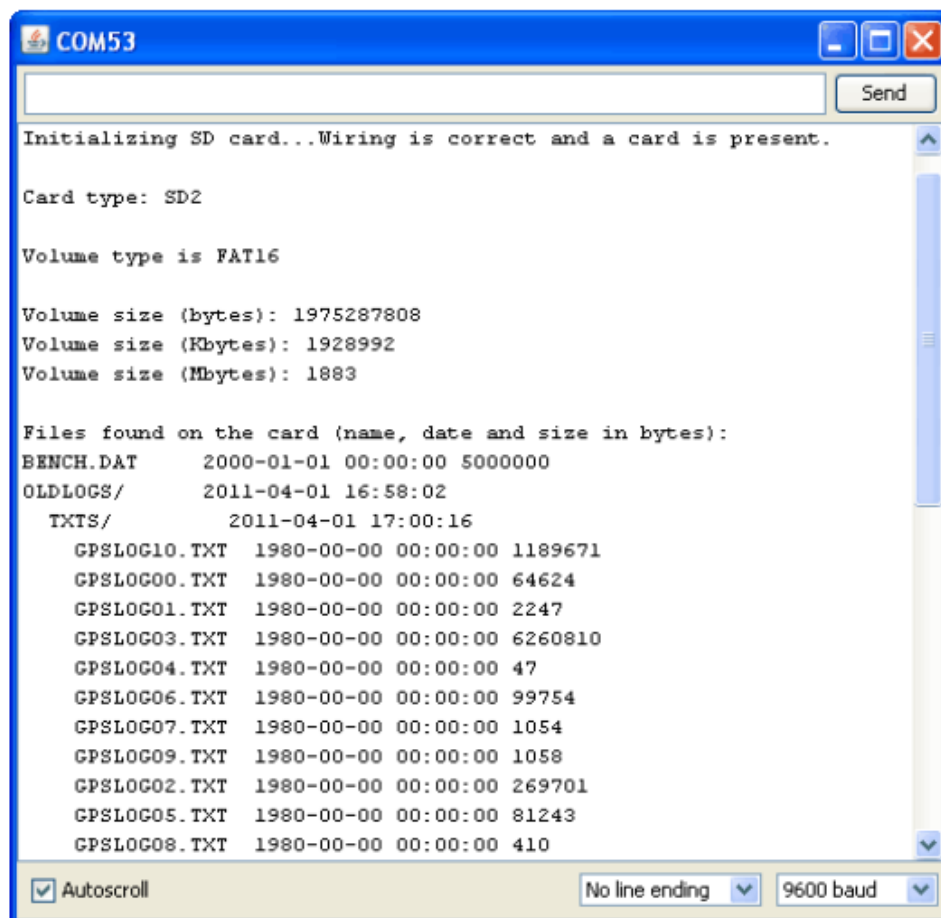


Fig. 25 Serial monitor showing the SD card initialization and setup

➡ 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.

```

1. #include<SD.h>
2.
3. File myFile;
4.
5. void setup()
6. {
7.   Serial.begin(9600);
8.   Serial.print("Initializing SD card...");
9.   // On the Ethernet Shield, CS is pin 4. It's set as an output by default.
10.  // Note that even if it's not used as the CS pin, the hardware SS pin
11.  // (10 on most Arduino boards, 53 on the Mega) must be left as an output
12.  // or the SD library functions will not work.
13.  pinMode(10, OUTPUT);
14.
15.  if (!SD.begin(10)) {
16.    Serial.println("initialization failed!");
17.    return;
18.  }
19.  Serial.println("initialization done.");
20.
21.  // open the file. note that only one file can be open at a time,
22.  // so we have to close this one before opening another.
23.  myFile = SD.open("test.txt", FILE_WRITE);
24.
25.  // if the file opened okay, write to it:
26.  if (myFile) {
27.    Serial.print("Writing to test.txt...");
28.    myFile.println("testing 1, 2, 3.");
29.    // close the file:
30.    myFile.close();
31.    Serial.println("done.");
32.  } else {
33.    // if the file didn't open, print an error:
34.    Serial.println("error opening test.txt");
35.  }
36. }
37.
38. void loop()
39. {
40.   // nothing happens after setup
41. }

```

When this code is uploaded it should results the information shown in Fig. 26.

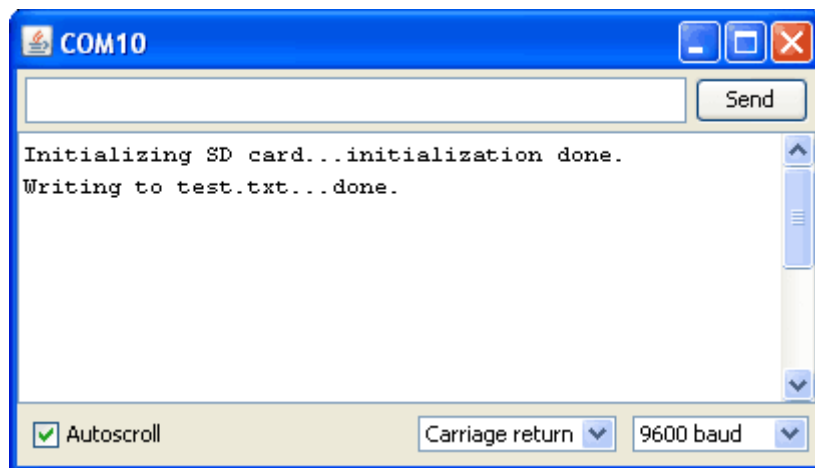


Fig. 26 Initializing SD card

A file in operating system can now be opened by inserting the card. One line will be seen for each time the sketch ran. That is to say, it **appends** to the file, not overwriting it.

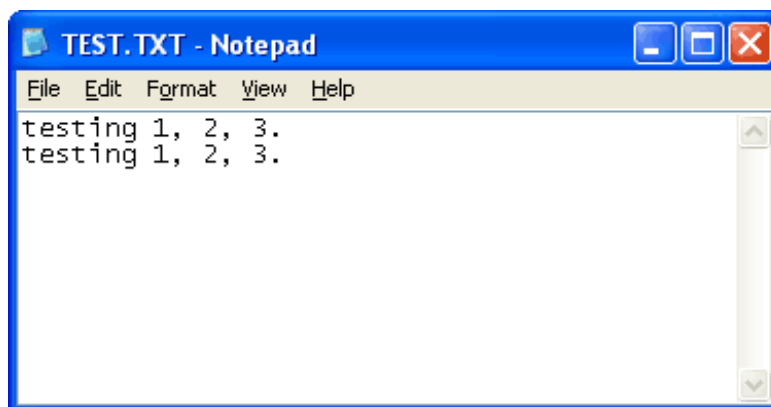


Fig. 27 testing the card

Some important points to note:

- Multiple files can be opened at a time, and write to each one as required.
- The print and println() can be used just like Serial objects, to write strings, variables, etc
- The file(s) must be closed i.e., close() when everything is done to make sure all the data is written permanently.
- The file can be opened in a directory i.e., /MyFiles/example.txt. **SD.open("/myfiles/example.txt") function can be called** and it will do the right thing.

✓ Reading from Files

Next task is to read from a file. It is very similar to writing, here **SD.open() function opens** the file but this time no argument is passed. This will keep it away from accidentally writing to it. **available () function can be called** which will let we know if there is data left to be read and **read () function** from the file, which will return the next byte.


```

1 #include<SD.h>

2
3 File myFile;
4
5 void setup()
6 {
7   Serial.begin(9600);
8   Serial.print("Initializing SD card...");
9   // On the Ethernet Shield, CS is pin 4. It's set as an output by default.
10  // Note that even if it's not used as the CS pin, the hardware SS pin
11  // (10 on most Arduino boards, 53 on the Mega) must be left as an output
12  // or the SD library functions will not work.
13  pinMode(10, OUTPUT);
14
15  if (!SD.begin(10)) {
16    Serial.println("initialization failed!");
17    return;
18  }
19  Serial.println("initialization done.");
20
21  // open the file for reading:
22  myFile = SD.open("test.txt");
23  if (myFile) {
24    Serial.println("test.txt:");
25
26    // read from the file until there's nothing else in it:
27    while (myFile.available()) {
28      Serial.write(myFile.read());
29    }
30    // close the file:
31    myFile.close();
32  } else {
33    // if the file didn't open, print an error:
34    Serial.println("error opening test.txt");
35  }
36
37  void loop() { }

```

➡ Recursively Listing/Reading Files

In the latest version of the SD library, the “recurse” can be done through a directory and call **openNextFile()** to get the next available file. These are not in alphabetical order but in order of creation.

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

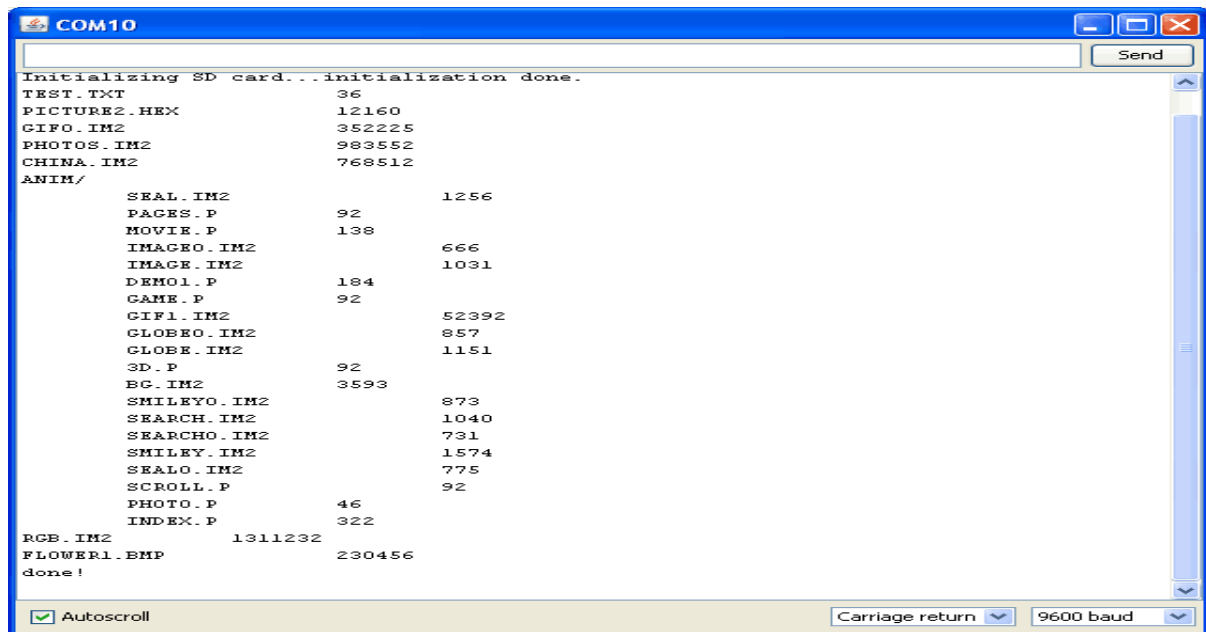


Fig. 28 Showing list of files

➡ Other Useful Functions

There are few other useful things that can be done with SD objects. Few of them are listed as under:

- For checking the existence of file use `SD.exists("filename.txt")` which will return true or false.
- A file can be deleted by calling `SD.remove("unwanted.txt")`. Be careful while using this option because there is no 'trash can' to pull it out of.
- A subdirectory can be created by calling `SD.mkdir("/mynewdir")` handy when there is a need to stuff files in a location. Also, there's a few useful things we can do with File objects:
- The `seek()` functions can be used to move the reading/writing pointer to a new location i.e., `seek(0)` will take pointer to the beginning of the file, which can be very handy! Likewise `position()` can be called accessing the current location in the file.
- The `size()` to get the number of bytes in the file.
- `Directory()` function can be used to determine if a file is a directory.
- Once there is a directory, all files in it can be accessed by `CallingopenNextFile()`.

Chapter: 07

Interfacing Arduino with Ethernet Shield

➡ 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.

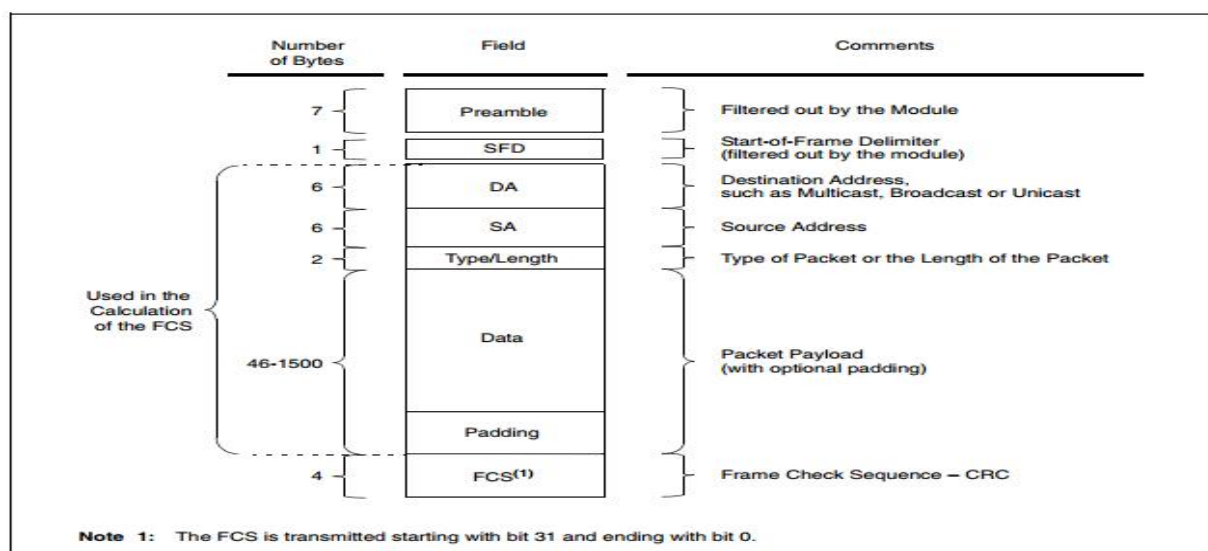


Fig. 29 IEEE 802.3 protocol stack

➡ 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.



Fig. 30 ENC28J60 Ethernet shield module

➡ ENC28J60 Pin-out I/O Description

The chart given in Fig. 31 is showing the complete details of I/O pins of ENC28J60 LAN module. For more info please refer ENC2J60 datasheet.

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 μ F and a minimum value of 1 μ F to ground, must be placed on this pin.
VSS	2	26	P	—	Ground reference.
CLKOUT	3	27	O	—	Programmable clock output pin. ⁽¹⁾
INT	4	28	O	—	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⁾
CS	9	5	I	ST	Chip select input pin for SPI interface. ^(3,4)
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.

Fig. 31 ENC28J60 pins description

➡ 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. Following registers need to be configured:

- 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 transfer 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.
- 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 having multiple SPI devices sharing the same MISO, MOSI, and CLK lines. To write code for a new SPI device following points need to be noted:

- Should Most Significant Bit (MSB) or Least Significant Bit (LSB) of the data be transferred 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 special attention need to be paid 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 Table 01.

Table 01: Modes of transmission

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 Table 02 displays mapping of SPI lines to pins of different Arduino boards.

Table 02: pin mapping of SPI and Arduino board

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

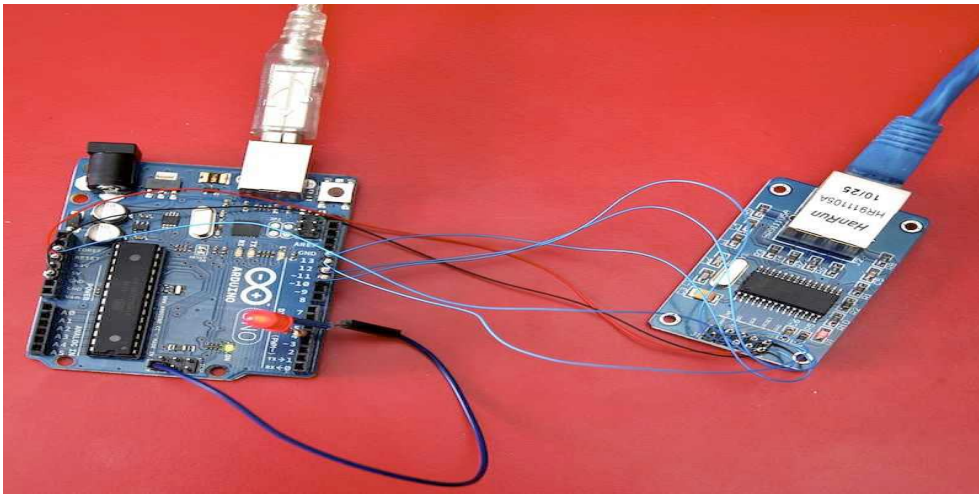


Fig. 32 Arduino interfaced with Ethernet shield

The Table 03 describing which pins on the Arduino should be connected to pins on ENC28J60 Ethernet Module.

Table 03: Pin mapping of Arduino board with Ethernet shield

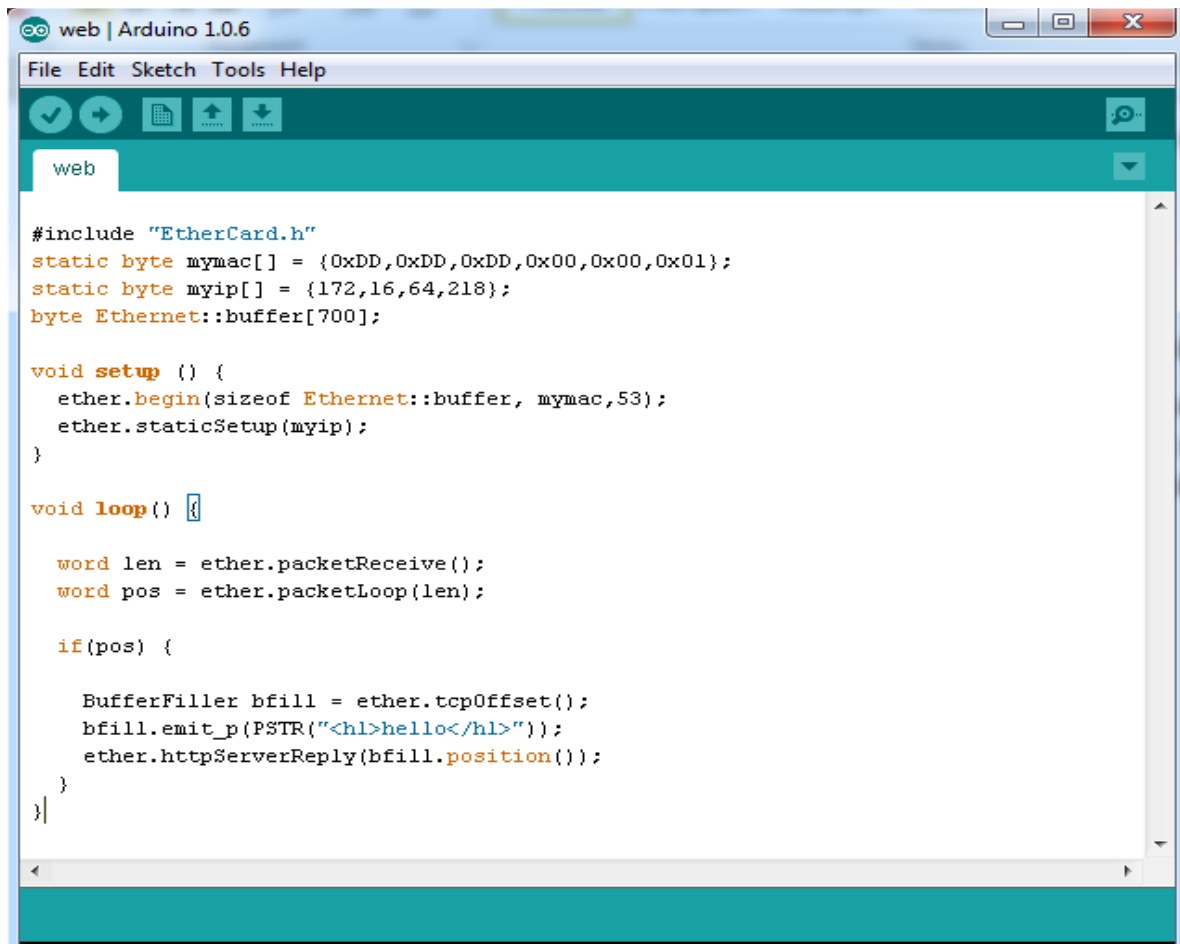
ENC28J60 module	Arduino Uno/Due	Arduino Mega
CS	D10	D53
SI	D11	D51
SO	D12	D50
SCK	D13	D52
RESET	RESET	RESET
INT	D2	D2
VCC	3V3	3V3
GND	GND	GND

➡ Ethernet Library Installation

There are several libraries of the Ethernet shield for Arduino i.e., UIPEthernet, Ether_28j60, and Ethercard. Download any of the libraries (Ethercard is recommended one) from internet and extract it to Arduino library folder.

✓ Example

An example code is shown in Fig. 33 in which a hello from Arduino is transferred through Ethernet shield to browser. Here you need to change myip [] address according to local LAN settings. All other parameters will remain same.



```

web | Arduino 1.0.6
File Edit Sketch Tools Help

web

#include "EthernetCard.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("<hl>hello</hl>"));
    ether.httpServerReply(bfill.position());
  }
}

```

Fig. 33 Simple sketch for transferring text from Arduino to browser

Chapter: 08

Exploring GPRS Module

➡ Introduction

This lab is basically based on GPRS/GSM based wireless communication using SIM908 module. This module requires AT commands for establishing voice call, SMS, HTTP, TCP/IP connections. AT Command Tester software will be used to learn these available AT interfaces in SIMCOM908 module.

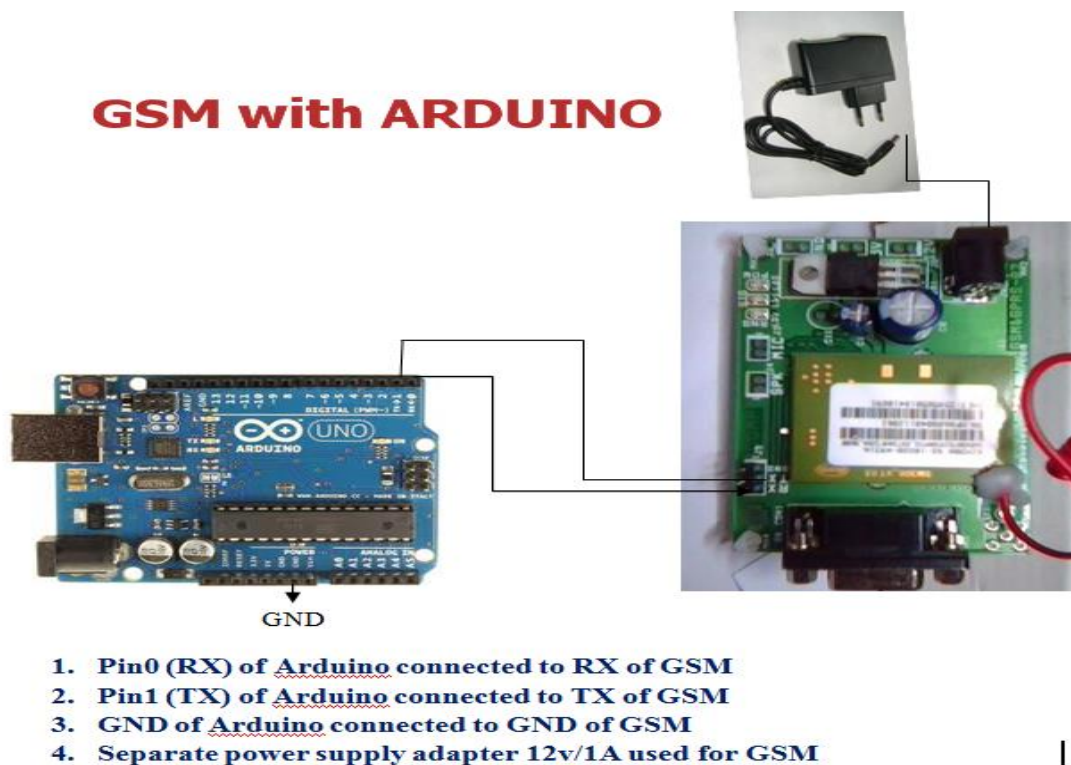


Fig. 34 Arduino interfaced with GPRS module

➡ Initialization Commands

Table 04: Commands for initialization of GPRS module

Command	Purpose
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

Table 05: Commands for setting up the voice call

Command	Purpose
AT+CREG?	Checking registration status...
ATD8586743398;	Dialing number 8586743398

➡ AT Command Sequence for Send SMS

Table 06: Commands for sending SMS

Command	Purpose
AT+CMGS="7608841145"	Send the SMS message
Eenter msg text +ending character ctrl+z	

➡ AT Command Sequence for HTTP

Table 07: Commands for http

Command	Purpose
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

Table 08: Commands for setting up TCP connection

Command	Purpose
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-alivectrl+z	