

Abstract

Train accidents are uncommon in India . But unfortunately when these accidents occur, people are often seriously injured or even killed. Accidents involving trains are often the result of mechanical failures and human error, and often it's a combination of both. There are a variety of different reasons that these railroad accidents occur, and some of these include:

- Accidental fire
- Train derailment
- Improper maintenance of the train tracks
- Faulty equipment
- Collision with another train
- Collision with a car , bus or truck trying to cross train tracks
- Collapsed bridges
- Faulty train crossings

The Indian Railways is the world's largest railway network covering an expanse of about 65,000 kms with 23 million passengers estimated to be travelling on it every day. Despite the magnitude of coverage of the service, the lack of basic health care facilities available to commuters is certainly a cause for concern." -India Together

CHAPTER 1

INTRODUCTION

1.1 Embedded Systems:

An embedded system is a special purpose computer system that is designed to perform very small sets of designated activities. Embedded systems date back as early as the late 1960s where they used to control electromechanical telephone switches. The first recognizable embedded system was the Apollo Guidance Computer developed by Charles Draper and his team. Later they found their way into the military, medical sciences and the aerospace and automobile industries.

Today they are widely used to serve various purposes like:

- Network equipment such as firewall, router, switch, and so on.
- Consumer equipment such as MP3 players, cell phones, PDAs, digital cameras, camcorders, home entertainment systems and so on.
- Household appliances such as microwaves, washing machines, televisions and so on.
- Mission-critical systems such as satellites and flight control.

The key factors that differentiate an embedded system from a desktop computer:

- They are cost sensitive.
- Most embedded systems have real time constraints.
- There are multitudes of CPU architectures such as ARM, MIPS, and PowerPC that are used in embedded systems. Application-specific processors are employed in embedded systems.
- Embedded Systems have and require very few resources in terms of ROM or other I/O devices as compared to a desktop computer.

1.2 Types of Setup

Embedded systems generally have a setup that includes a host which is generally a personal computer, and a target that actually executes all the embedded applications. The various types of host/ desktop architectures that are used in embedded systems are:

Linked Setup:

In this setup, the target and the host are permanently linked together using a physical cable. This link is typically a serial cable or an Ethernet link. The main property of this setup is

that no physical hardware storage device is being transferred between the target and the host. The host contains the cross-platform development environment while the target contains an appropriate bootloader, a functional kernel, and a minimal root filesystem.

Removable Storage Setup:

In the removable setup, there are no direct physical links between the host and the target. Instead, a storage device is written by the host, is then transferred into the target, and is used to boot the device. The host contains the cross-platform development environment. The target, however, contains only a minimal bootloader. The rest of the components are stored on a removable storage media, such as a CompactFlash IDE device, MMC Card, or any other type of removable storage device.

Standalone Setup:

The target is a self-contained development system and includes all the required software to boot, operate, and develop additional software. In essence, this setup is similar to an actual workstation, except the underlying hardware is not a conventional workstation but rather the embedded system itself. This one does not require any cross-platform development environment, since all development tools run in their native environments. Furthermore, it does not require any transfer between the target and the host, because all the required storage is local to the target.

1.3 Operating Systems

In an embedded system, when there is only a single task that is to be performed, then only a binary is to be loaded into the target controller and is to be executed. However, when there are multiple tasks to be executed or multiple events to be handled, then there has to be a program that handles and prioritizes these events. This program is the Operating System (OS), which one is very familiar with, in desktop PCs.

Various Operating Systems:

Embedded Operating Systems are classified into two categories:

1.3.1 Real-time Operating Systems (RTOS) :

Real Time Operating Systems are those which guarantee responses to each event within a defined amount of time. This type of operating system is mainly used by time-critical applications such as measurement and control systems. Some commonly used RTOS for embedded systems are: VxWorks, OS-9, Symbian,

1.3.2 Non-Real-time Operating Systems:

Non-Real Time Operating Systems do not guarantee defined response times. These systems are mostly used if multiple applications are needed. Windows CE and PalmOS are examples for such embedded operating systems.

Why Linux?

There are a wide range of motivations for choosing Linux over a traditional embedded OS.

The following are the criteria due to which Linux is preferred:

I. Quality and Reliability of Code:

Quality and reliability are subjective measures of the level of confidence in the code that comprises software such as the kernel and the applications that are provided by distributions. Some properties that professional programmers expect from a “quality” code are modularity and structure, readability, extensibility and configurability. “Reliable” code should have features like predictability, error recovery and longevity. Most programmers agree that the Linux kernel and other projects used in a Linux system fit this description of quality and reliability. The reason is the open source development model, which invites many parties to contribute to projects, identify existing problems, debate possible solutions, and fix problems effectively.

II. Availability of Code:

Code availability relates to the fact that the Linux source code and all build tools are available without any access restrictions. The most important Linux components, including the kernel itself, are distributed under the GNU General Public License (GPL). Access to these components’ source code is therefore compulsory (at least to those users who have purchased any system running GPL-based software, and they have the right to redistribute once they obtain the source in any case). Code availability has implications for standardization and commoditization of components, too. Since it is possible to build Linux systems based entirely upon software for which source is available, there is a lot to be gained from adopting standardized embedded software platforms.

1.4 Hardware Support:

Broad hardware support means that Linux supports different types of hardware platforms and devices. Although a number of vendors still do not provide Linux drivers, considerable

progress has been made and more is expected. Because a large number of drivers are maintained by the Linux community itself, you can confidently use hardware components without fear that the vendor may one day discontinue driver support for that product line. Linux also provides support for dozens of hardware architectures. No other OS provides this level of portability.

Typical architecture of an Embedded Linux System

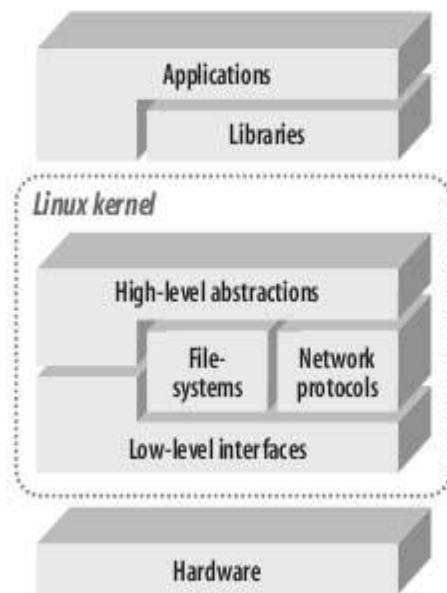


Fig- 1.1 hardware support

1.4.1 Hardware

Linux normally requires at least a 32-bit CPU containing a memory management unit (MMU). A sufficient amount of RAM must be available to accommodate the system. Minimal I/O capabilities are required if any development is to be carried out on the target with reasonable debugging facilities. The kernel must be able to load a root filesystem through some form of permanent storage, or access it over a network.

1.4.2 Linux Kernel:

Immediately above the hardware sits the kernel, the core component of the operating system. Its purpose is to manage the hardware in a coherent manner while providing familiar high-level abstractions to user-level software. It is expected that applications using the APIs

provided by a kernel will be portable among the various architectures supported by this kernel with little or no changes. The low-level interfaces are specific to the hardware configuration on which the kernel runs and provide for the direct control of hardware resources using a hardware-independent API. Higher-level components provide the abstractions common to all UNIX systems, including processes, files, sockets, and signals. Since the low-level APIs provided by the kernel are common among different architectures, the code implementing the higher-level abstractions is almost constant, regardless of the underlying architecture. Between these two levels of abstraction, the kernel sometimes needs what could be called interpretation components to understand and interact with structured data coming from or going to certain devices. Filesystem types and networking protocols are prime examples of sources of structured data the kernel needs to understand and interact with in order to provide access to data going to and coming from these sources.

1.4.3 Applications and Libraries:

Applications do not directly operate above the kernel, but rely on libraries and special system daemons to provide familiar APIs and abstract services that interact with the kernel on the application's behalf to obtain the desired functionality. The main library used by most Linux applications is the GNU C library, glibc. For Embedded Linux systems, substitutes to this library that are much less in size than glibc are preferred.

Embedded systems software

One of the key elements of any embedded system is the software that is used to run the microcontroller.

There is a variety of ways that this can be written:

- ***Machine code:*** Machine code is the most basic code that is used for the processor unit. The code is normally in hex code and provides the basic instructions for each operation of the processor. This form of code is rarely used for embedded systems these days.
- ***Programming language:*** Writing machine code is very laborious and time consuming. It is difficult to understand and debug. To overcome this, high level programming languages are often used. Languages including C, C++, etc are commonly used.

Embedded systems design tools

Many embedded systems are complicated and require large levels of software for them to operate.

Developing this software can be timing consuming, and it has to be very accurate for the embedded system to operate correctly. Coding in embedded systems is one of the main areas where faults occur.

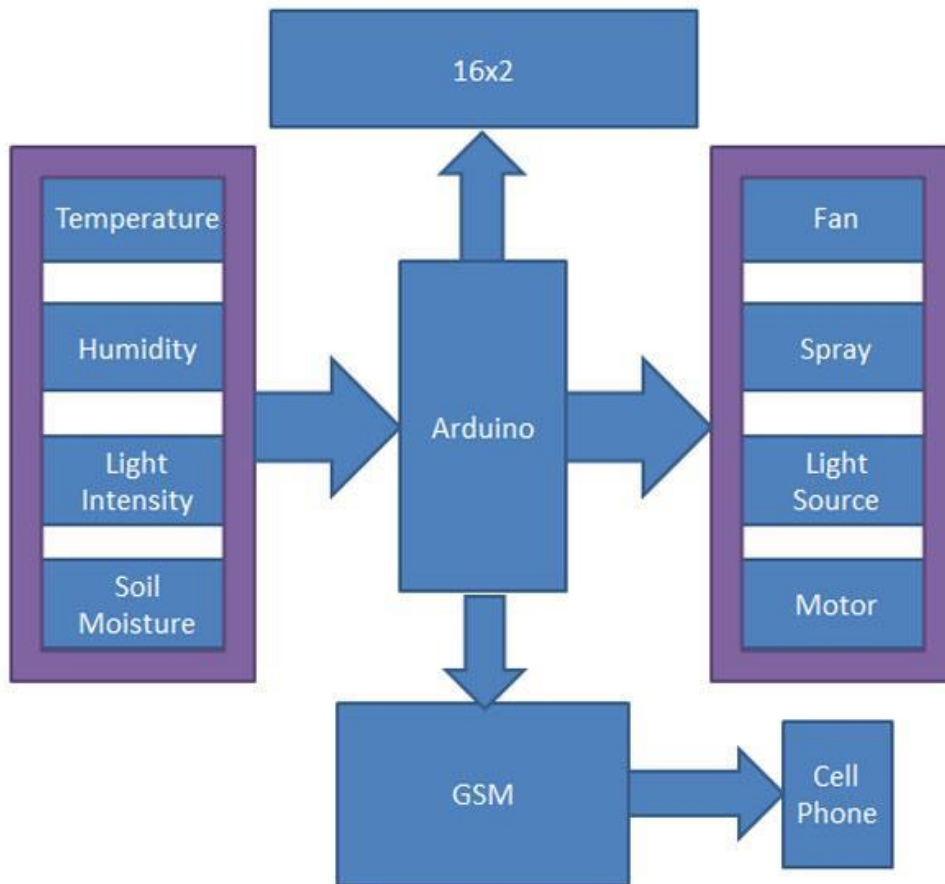
To help simplify the process, software development tools are normally used. These help the software developer to programmed more quickly, and also more accurately..

Typically the software is written in a high level and then compiled down into a form that is contained within the non-volatile memory of the system. This is normally referred to as the firmware.

CHAPTER 2

LITERATURE SURVEY

2.1 Block Diagram:



2.2 Regulated power supply:

2.3 Control unit:

2.4 RF Module:

2.5 Sensor:

2.6 Serial & Parallel Conversation:

2.7 IOT System:

2.8 Ethernet Shield:

CHAPTER 3

AURDINO UNO

Arduino/Genuino Uno is a microcontroller board based on the ATmega328P (datasheet). It has 14 digital input/output pins (of which 6 can be used as PWM outputs), 6 analog inputs, a 16 MHz quartz crystal, 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..

"Uno" means one in Italian and was chosen to mark the release of Arduino Software (IDE) 1.0. The Uno board and version 1.0 of Arduino Software (IDE) were the reference versions of Arduino, now evolved to newer releases. The Uno board is the first in a series of USB Arduino boards, and the reference model for the Arduino platform.

Let's face it, there are a lot of different Arduino boards out there. How do you decide which one you need for your project? In this tutorial, we'll take a look at the diverse world of Arduino boards. We'll begin with a tabular overview of the features each board has. Then we'll delve deeper into each board, examining the pros, cons, and example use-cases.

Arduino is an open-source electronics prototyping platform based on flexible, easy-to-use hardware and software. It's intended for artists, designers, hobbyists, and anyone interested in creating interactive objects or environments. Or more simply, you load on some code and it can read sensors, perform actions based on inputs from buttons, control motors, and accept shields to further expand it's capabilities. Really, you can do almost anything.

All Arduino boards have one thing in common: they are programmed through the Arduino IDE. This is the software that allows you to write and upload code. Beyond that, there can be a lot of differences. The number of inputs and outputs (how many sensors, LEDs, and buttons you can use on a single board), speed, operating voltage, and form factor are just a few of the variables. Some boards are designed to be embedded and have no programming interface (hardware) which you would need to buy separately. Some can run directly from a 3.7V battery, others need at least 5V. Check the chart on the next page to find the right Arduino for your project.

Arduino is an open-source electronics platform based on easy-to-use hardware and software. Arduino boards are able to read inputs - light on a sensor, a finger on a button, or a Twitter message - and turn it into an output - activating a motor, turning on an LED, publishing something online. You can tell your board what to do by sending a set of instructions to the microcontroller on the board. To do so you use the Arduino programming language (based on Wiring), and the Arduino Software (IDE), based on Processing.

Over the years Arduino has been the brain of thousands of projects, from everyday objects to complex scientific instruments. A worldwide community of makers - students, hobbyists, artists, programmers, and professionals - has gathered around this open-source platform, their contributions have added up to an incredible amount of accessible knowledge that can be of great help to novices and experts alike.

Arduino was born at the Ivrea Interaction Design Institute as an easy tool for fast prototyping, aimed at students without a background in electronics and programming. As soon as it reached a wider community, the Arduino board started changing to adapt to new needs and challenges, differentiating its offer from simple 8-bit boards to products for IoT applications, wearable, 3D printing, and embedded environments. All Arduino boards are completely open-source, empowering users to build them independently and eventually adapt them to their particular needs. The software, too, is open-source, and it is growing through the contributions of users worldwide.

Thanks to its simple and accessible user experience, Arduino has been used in thousands of different projects and applications. The Arduino software is easy-to-use for beginners, yet flexible enough for advanced users. It runs on Mac, Windows, and Linux. Teachers and students use it to build low cost scientific instruments, to prove chemistry and physics principles, or to get started with programming and robotics. Designers and architects build interactive prototypes, musicians and artists use it for installations and to experiment with new musical instruments. Makers, of course, use it to build many of the projects exhibited at the Maker Faire, for example. Arduino is a key tool to learn new things. Anyone - children, hobbyists, artists, programmers - can start tinkering just following the step by step instructions of a kit, or sharing ideas online with other members of the Arduino community.

There are many other microcontrollers and microcontroller platforms available for physical computing. Parallax Basic Stamp, Netmedia's BX-24, Phidgets, MIT's Handyboard, and many others offer similar functionality. All of these tools take the messy details of microcontroller programming and wrap it up in an easy-to-use package. Arduino also simplifies the process of working with microcontrollers, but it offers some advantage for teachers, students, and interested amateurs over other systems:

Inexpensive - Arduino boards are relatively inexpensive compared to other microcontroller platforms. The least expensive version of the Arduino module can be assembled by hand, and even the pre-assembled Arduino modules cost less than \$50

Cross-platform - The Arduino Software (IDE) runs on Windows, Macintosh OSX, and Linux operating systems. Most microcontroller systems are limited to Windows.

Simple, clear programming environment - The Arduino Software (IDE) is easy-to-use for beginners, yet flexible enough for advanced users to take advantage of as well. For teachers, it's

conveniently based on the Processing programming environment, so students learning to program in that environment will be familiar with how the Arduino IDE works.

Open source and extensible software - The Arduino software is published as open source tools, available for extension by experienced programmers. The language can be expanded through C++ libraries, and people wanting to understand the technical details can make the leap from Arduino to the AVR C programming language on which it's based. Similarly, you can add AVR-C code directly into your Arduino programs if you want to.

Open source and extensible hardware - The plans of the Arduino boards are published under a Creative Commons license, so experienced circuit designers can make their own version of the module, extending it and improving it. Even relatively inexperienced users can build the breadboard version of the module in order to understand how it works and save money.

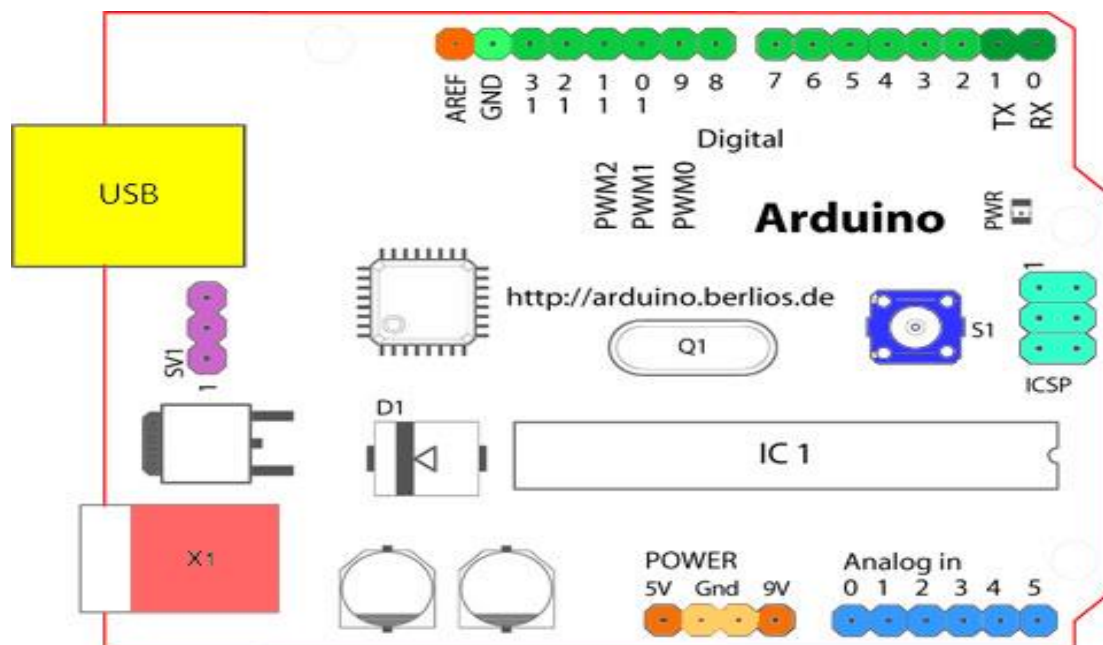
How do I use Arduino?

See the getting started guide. If you are looking for inspiration you can find a great variety of Tutorials on Arduino Project Hub.

The text of the Arduino getting started guide is licensed under a Creative Commons Attribution-ShareAlike 3.0 License. Code samples in the guide are released into the public domain.

Introduction to the Arduino Board

Looking at the board from the top down, this is an outline of what you will see (parts of the board you might interact with in the course of normal use are highlighted):



Starting clockwise from the top center:

- Analog Reference pin (orange)
- Digital Ground (light green)
- Digital Pins 2-13 (green)
- Digital Pins 0-1/Serial In/Out - TX/RX (dark green) - *These pins cannot be used for digital i/o (digitalRead and digitalWrite) if you are also using serial communication (e.g. Serial.begin).*
- Reset Button - S1 (dark blue)
- In-circuit Serial Programmer (blue-green)
- Analog In Pins 0-5 (light blue)
- Power and Ground Pins (power: orange, grounds: light orange)
- External Power Supply In (9-12VDC) - X1 (pink)
- Toggles External Power and USB Power (place jumper on two pins closest to desired supply) - SV1 (purple)
- USB (used for uploading sketches to the board and for serial communication between the board and the computer; can be used to power the board) (yellow)

Microcontrollers

Digital Pins

In addition to the specific functions listed below, the digital pins on an Arduino board can be used for general purpose input and output via the `pinMode()`, `digitalRead()`, and `digitalWrite()` commands. Each pin has an internal pull-up resistor which can be turned on and off using `digitalWrite()` (w/ a value of HIGH or LOW, respectively) when the pin is configured as an input. The maximum current per pin is 40 mA.

- Serial: 0 (RX) and 1 (TX). Used to receive (RX) and transmit (TX) TTL serial data. On the Arduino Diecimila, these pins are connected to the corresponding pins of the FTDI USB-to-TTL Serial chip. On the Arduino BT, they are connected to the corresponding pins of the WT11 Bluetooth module. On the Arduino Mini and LilyPad Arduino, they are intended for use with an external TTL serial module (e.g. the Mini-USB Adapter).
- External Interrupts: 2 and 3. These pins can be configured to trigger an interrupt on a low value, a rising or falling edge, or a change in value. See the `attachInterrupt()` function for details.
- PWM: 3, 5, 6, 9, 10, and 11. Provide 8-bit PWM output with the `analogWrite()` function. On boards with an ATmega8, PWM output is available only on pins 9, 10, and 11.
- BT Reset: 7. (Arduino BT-only) Connected to the reset line of the bluetooth module.
- SPI: 10 (SS), 11 (MOSI), 12 (MISO), 13 (SCK). These pins support SPI communication, which, although provided by the underlying hardware, is not currently included in the Arduino language.

- LED: 13. On the Diecimila and LilyPad, there is a built-in LED connected to digital pin 13. When the pin is HIGH value, the LED is on, when the pin is LOW, it's off.

Analog Pins

In addition to the specific functions listed below, the analog input pins support 10-bit analog-to-digital conversion (ADC) using the `analogRead()` function. Most of the analog inputs can also be used as digital pins: analog input 0 as digital pin 14 through analog input 5 as digital pin 19. Analog inputs 6 and 7 (present on the Mini and BT) cannot be used as digital pins.

- I²C: 4 (SDA) and 5 (SCL). Support I²C (TWI) communication using the Wire library (documentation on the Wiring website).

Power Pins

- VIN (sometimes labelled "9V"). The input voltage to the Arduino board when it's using an external power source (as opposed to 5 volts from the USB connection or other regulated power source). You can supply voltage through this pin, or, if supplying voltage via the power jack, access it through this pin. Note that different boards accept different input voltages ranges, please see the documentation for your board. Also note that the LilyPad has no VIN pin and accepts only a regulated input.
- 5V. The regulated power supply used to power the microcontroller and other components on the board. This can come either from VIN via an on-board regulator, or be supplied by USB or another regulated 5V supply.
- 3V3. (Diecimila-only) A 3.3 volt supply generated by the on-board FTDI chip.
- GND. Ground pins.

Other Pins

- AREF. Reference voltage for the analog inputs. Used with `analogReference()`.
- Reset. (Diecimila-only) Bring this line LOW to reset the microcontroller. Typically used to add a reset button to shields which block the one on the board.

Reference Home

Corrections, suggestions, and new documentation should be posted to the Forum.

The text of the Arduino reference is licensed under a Creative Commons Attribution-ShareAlike 3.0 License. Code samples in the reference are released into the public domain.

Glossary of Terms:

Microcontroller (MCU): The microcontroller is the heart (or, more appropriately, the brain) of the Arduino board. The Arduino development board is based on AVR microcontrollers of different types, each of which have different functions and features.

Input Voltage: This is the suggested input voltage range for the board. The board may be rated for a slightly higher maximum voltage, but this is the safe operating range. A handy thing to keep in mind is that many of the Li-Po batteries that we carry are 3.7V, meaning that any board with an input voltage including 3.7V can be powered directly from one of our Li-Po battery packs.

System Voltage: This is the system voltage of the board, i.e. the voltage at which the microcontroller is actually running. This is an important factor for shield-compatibility since the logic level is now 3.3V instead of 5V. You always want to be sure that whatever outside system with which you're trying to communicate is able to match the logic level of your controller.

Clock Speed: This is the operating frequency of the microcontroller and is related to the speed at which it can execute commands. Although there are rare exceptions, most ATmega microcontrollers running at 3V will be clocked at 8MHz, whereas most running at 5V will be clocked at 16MHz. The clock speed of the Arduino can be divided down for power savings with a few tricks if you know what you're doing.

Digital I/O: This is the number of digital input/output (I/O) pins that are broken out on the Arduino board. Each of these can be configured as either an input or an output. Some are capable of PWM, and some double as serial communication pins.

Analog Inputs: This is the number of analog input pins that are available on the Arduino board. Analog pins are labeled "A" followed by their number, they allow you to read analog values using the analog-to-digital converter (ADC) in the ATmega chip. Analog inputs can also be configured as more digital I/O if you need it!

PWM: This is the number of digital I/O pins that are capable of producing a Pulse-width modulation. (PWM) signal. A PWM signal is like an analog output; it allows your Arduino to "fake" an analog voltage between zero and the system voltage.

UART: This is the number of separate serial communication lines your Arduino board can support. On most Arduino boards, digital I/O pins 0&1 double as your serial send and receive pins and are shared with the serial programming port. Some Arduino boards have multiple UARTs and

can support multiple serial ports at once. All Arduino boards have at least one UART for programming, but some aren't broken out to pins that are accessible.

Flash Space: This is the amount of program memory that the chip has available for you to store your sketch. Not all of this memory is available as a very small portion is taken up by the bootloader (usually between 0.5 and 2KB).

Programming Interface: This is how you hook up the Arduino board to your computer for programming. Some boards have a USB jack on-board so that all you need to do is plug them into a USB cable. Others have a header available so that you can plug in an FTDI Basic breakout or FTDI Cable. Other boards, like the Mini, break out the serial pins for programming but aren't pin-compatible with the FTDI header. Any Arduino board that has a USB jack on-board also has some other hardware that enables the serial to USB conversion. Some boards, however, don't need additional hardware because their microcontrollers have built-in support for USB.

ATmega328 Boards

The ATmega328 (and the ATmega168 before that, and ATmega8 before that,...) is a staple of the Arduino platform. 32kB of flash (program space), up to 23 I/Os – eight of which can be analog inputs – operating frequencies of up to 20 MHz. None of its specifications are flashy, but this is still a *solid* 8-bit microcontroller. For many electronics projects, what the 328 provides is still more than enough.

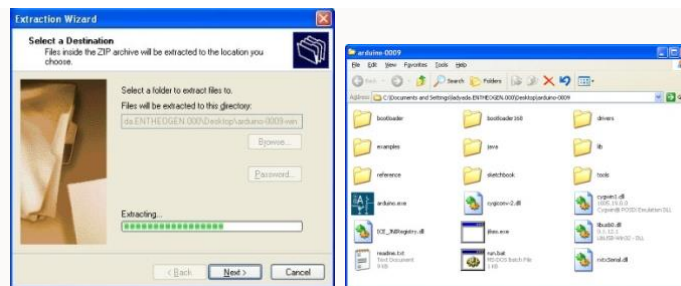
The Arduino boards on this page all feature the ATmega328 as their main MCU brain. The microcontroller alone makes every board on this page nearly identical in terms of I/O count and memory. Their differences stem from things like programming interfaces, form factors, and operating voltages.

The Main Event: Arduino Uno

The Arduino Uno is the “stock” Arduino. It's what we compare every, other, Arduino-compatible board to. If you're just getting into Arduino, **this is the board to start with.**

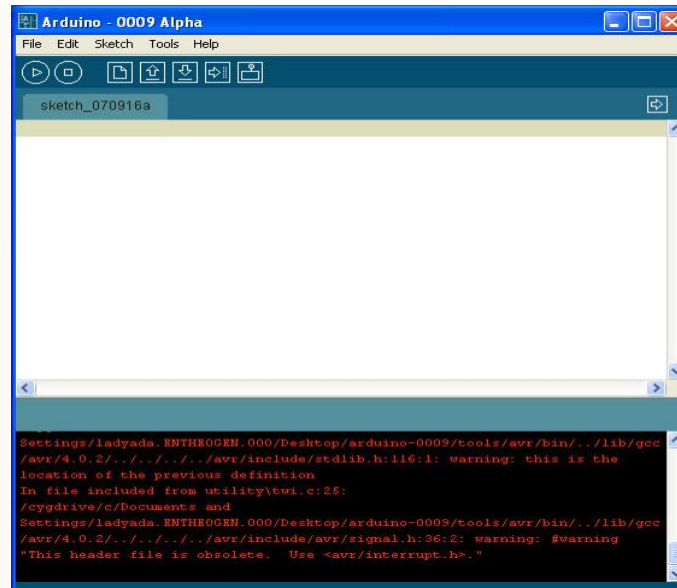
The Uno comes in two flavors, through-hole and SMD, which use either a through-hole or surface-mount ATmega328. The through-hole version (pictured above) is nice because you can take the chip out and swap in a new one (in case the magic, blue smoke is released), but the SMD version has the potential to be more readily available (PTH chips are increasingly being phased out of existence).

The Arduino Uno can be powered through either the USB interface, or an external barrel jack. To connect it to a computer you'll need a type-B-to-A USB cable (like the USB connector on most printers).



Windows

Mac OS X



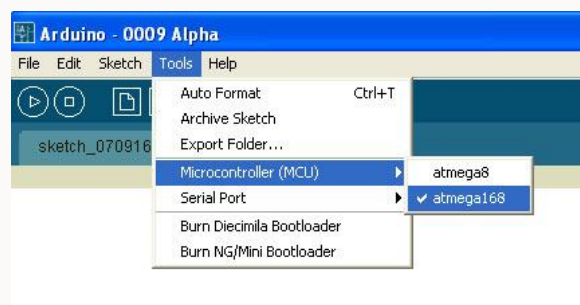
I think I get the red error text shown because I already have Arduino installed. Either way, it isn't a problem if you do or don't see it.

Select chip

The first step is to configure the Arduino software for the correct chip. Almost all Arduinos use the ATmega168, but there's a chance you have an ATmega8. Look for the chip on the Arduino that looks like this:



If the text says ATMEGA8-16P then you have an **atmega8** chip. If the text says ATMEGA168-20P then you have an **atmega168** chip. If it says "ATMEGA328P-20P" you have an **atmega328p** chip



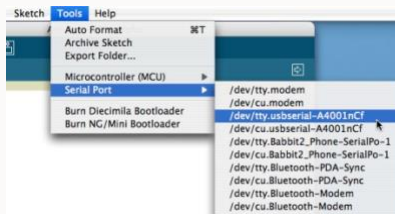
Make sure the correct chip is selected (this picture is really old, will be fixed soon). This preference is saved so you only have to set it once, the program will remember next time it's run.

Select port

Next, its time to configure the Serial Port (also known as the COM Port). Go back to [lesson 0](#) to remind yourself of which port it is. On a PC it will probably be something like **COM3** or **COM4**. On a Mac it will be something like **ttty.usbserial-xxxxx**



Windows port selection



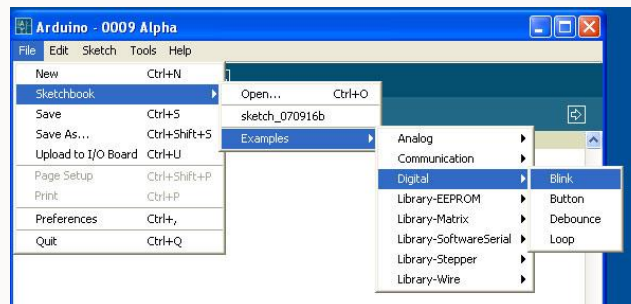
Mac port selection

This preference is saved so you only have to set it once, the program will remember next time it's run.

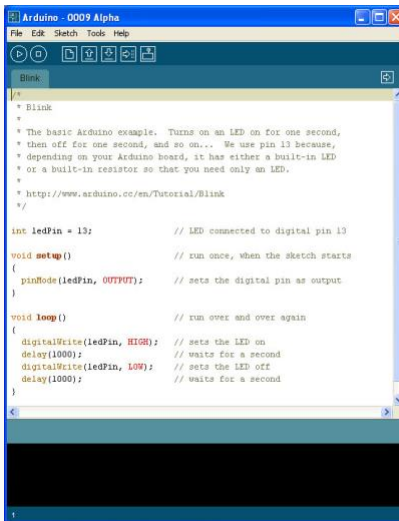
However, if you have multiple Arduino's, they may be assigned difference COM ports. So every time you plug in a new Arduino, double check that the correct port is selected.

Open blink sketch

Sketches are little scripts that you can send to the Arduino to tell it how to act. Let's open up an **Example Sketch**. Go to the **File menu -> Sketchbook -> Examples -> Digital -> Blink**



The window should now look like this, with a bunch of text in the formerly empty white space and the tab **Blink** above it

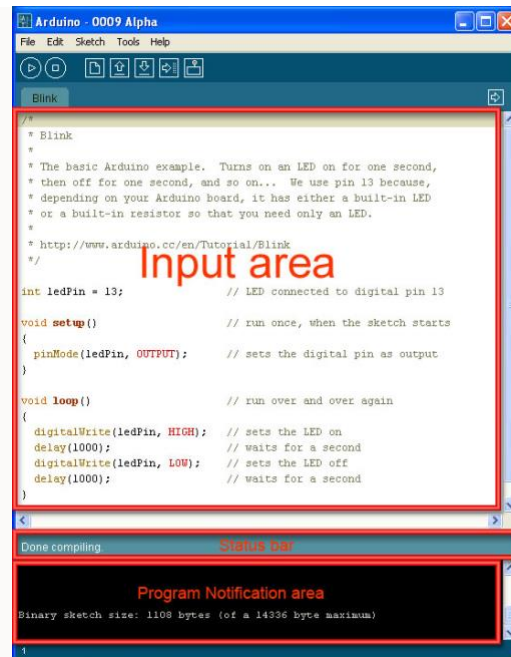


Verify / Compile

The first step to getting a **Sketch** ready for transfer over to the arduino is to **Verify/Compile** it. That means check it over for mistakes (sort of like editing) and then translate it into an application that is compatible with the Arduino hardware.

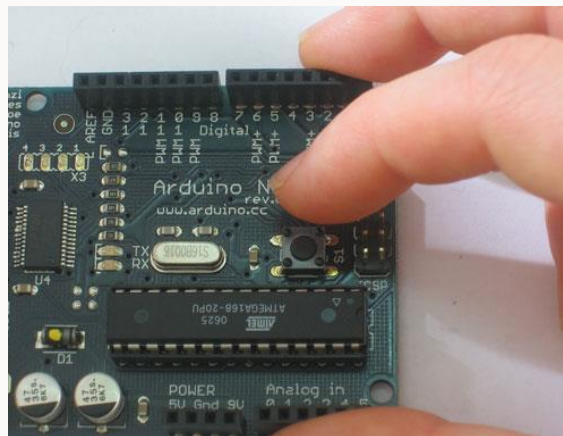


After a few seconds, you should see the message **Done compiling.** in the **Status Bar** and **Binary Sketch Size:** in the **Notification area**. This means the sketch was well-written and is ready for uploading to the Arduino board!



Reset (NG only)

To tell the Arduino that it should prepare itself for a new Sketch upload, you must reset the board. Diecimila Arduino's have built-in auto-reset capability, so you don't need to do anything. Older Arduinos, such as NG, must be manually reset before uploading a sketch. To do that simply press the black button on the right hand side of the board, shown here.

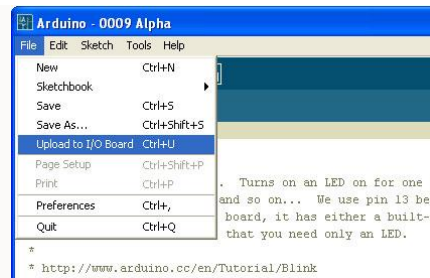


Upload

Now it's time to upload. Make sure the Arduino is plugged in, the green light is on and the correct Serial Port is selected.

If you have an NG Arduino, press the **Reset Button** now, just before you select the **Upload** menu item.

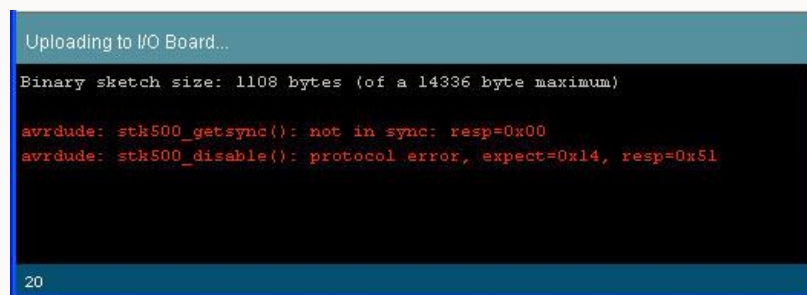
Select **Upload to I/O Board** from the **File** menu



After a few seconds you should get this screen, with the message **Done uploading.** in the status bar.



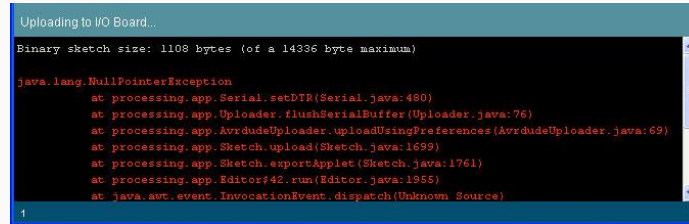
If you get the following error message "**avrdude: stk500_getsync(): not in sync: resp=0x00**" that means that the Arduino is not responding



Then check the following:

- If you have a NG Arduino, did you press reset just before selecting **Upload** menu item?
- Is the correct Serial Port selected?
- Is the correct driver installed?
- Is the chip inserted into the Arduino properly? (If you built your own arduino or have burned the bootloader on yourself)
- Does the chip have the correct bootloader on it? (If you built your own arduino or have burned the bootloader on yourself)

If you get the following error message:

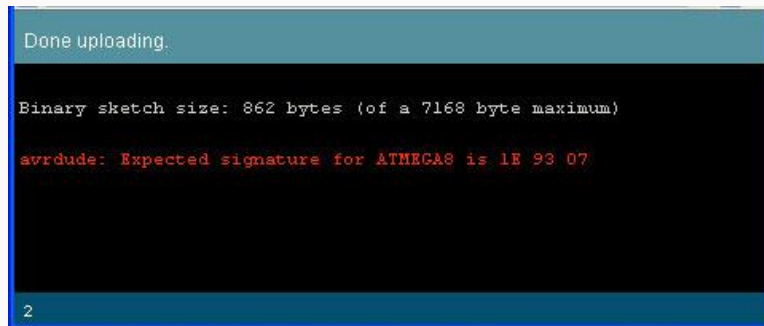


```
Uploading to I/O Board...
Binary sketch size: 1108 bytes (of a 14336 byte maximum)

java.lang.NullPointerException
    at processing.app.Serial.setDTR(Serial.java:480)
    at processing.app.Uploader.flushSerialBuffer(Uploader.java:76)
    at processing.app.AvrDudeUploader.uploadUsingPreferences(AvrDudeUploader.java:69)
    at processing.app.Sketch.upload(Sketch.java:1699)
    at processing.app.Sketch.exportApplet(Sketch.java:1761)
    at processing.app.Editor42.run(Editor.java:1955)
    at java.awt.event.InvocationEvent.dispatch(Unknown Source)
```

It means you don't have a serial port selected, go back and verify that the correct driver is installed ([lesson 0](#)) and that you have the correct serial port selected in the menu.

If you get the following error **Expected signature for ATMEGA**



```
Done uploading.

Binary sketch size: 862 bytes (of a 7168 byte maximum)

avrdude: Expected signature for ATMEGA8 is 1E 93 07
```

Then you have either the incorrect chip selected in the **Tools** menu or the wrong bootloader burned onto the chip

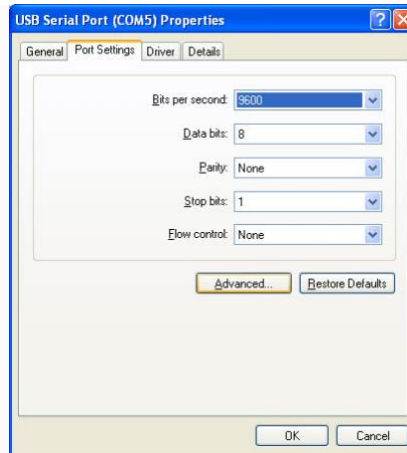
If you get the following error: **can't open device "COM10": The system cannot find the file specified** (under Windows, COM port value may vary)



```
Uploading to I/O Board...
Binary sketch size: 1108 bytes (of a 14336 byte maximum)

avrdude: ser_open(): can't open device "COM21": The system cannot find the
file specified.
```

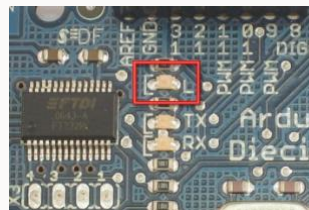
It means that you have too many COM ports (maybe you've got 9 Arduinos?) You should make sure that the port is numbered as low as possible. You can use a program like [FTClean](#) to clear out old COM ports you aren't using anymore. Once you've cleaned out the ports, you'll have to reinstall the driver again (see lesson 0). Alternately, if you're sure that the ports are not used for something else but are left over from other USB devices, you can simply change the COM port using the **Device Manager**. Select the USB device in the Device Manager, right click and select **Properties**



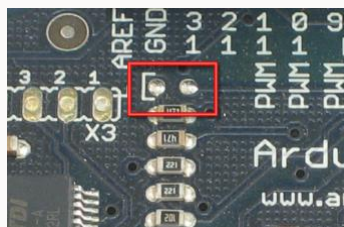
Insert LED (NG Arduinos)

Some older Arduinos don't have a built in LED, its easy to tell if yours does or not

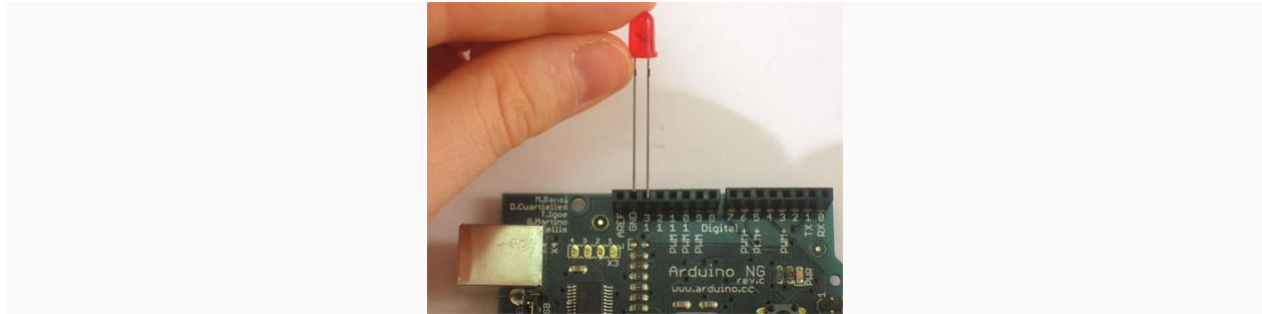
If you have a Diecimila or other Arduino with a built in LED you will see a translucent part as shown



If you have an NG rev C or other Arduino without an LED, the translucent part will not be there, and instead you will see two silver dots



If you don't have an LED, you'll need to add your own. Any LED will do, as long as it has two legs and kinda looks like the one shown here. LEDs are **directional** components. That means if you put it in backwards it will not work! To help you put the LED in right, the LED factory cuts the legs at different lengths. The longer leg goes in the hole marked **13** and the shorter one goes in the hole marked **GND**



Understanding Arduino UNO Hardware Design

This article explains how Arduino works from an electronic design perspective.

Most articles explain the software of Arduinos. However, understanding hardware design helps you to make the next step in the Arduino journey. A good grasp of the electronic design of your Arduino hardware will help you learn how to embed an Arduino in the design of a final product, including what to keep and what to omit from your original design.

Components Overview

The PCB design of the Arduino UNO uses SMD (Surface Mount Device) components. I entered the SMD world years ago when I dug into Arduino PCB design while I was a part of a team redesigning a DIY clone for Arduino UNO.

Integrated circuits use standardized packages, and there are families for packages.

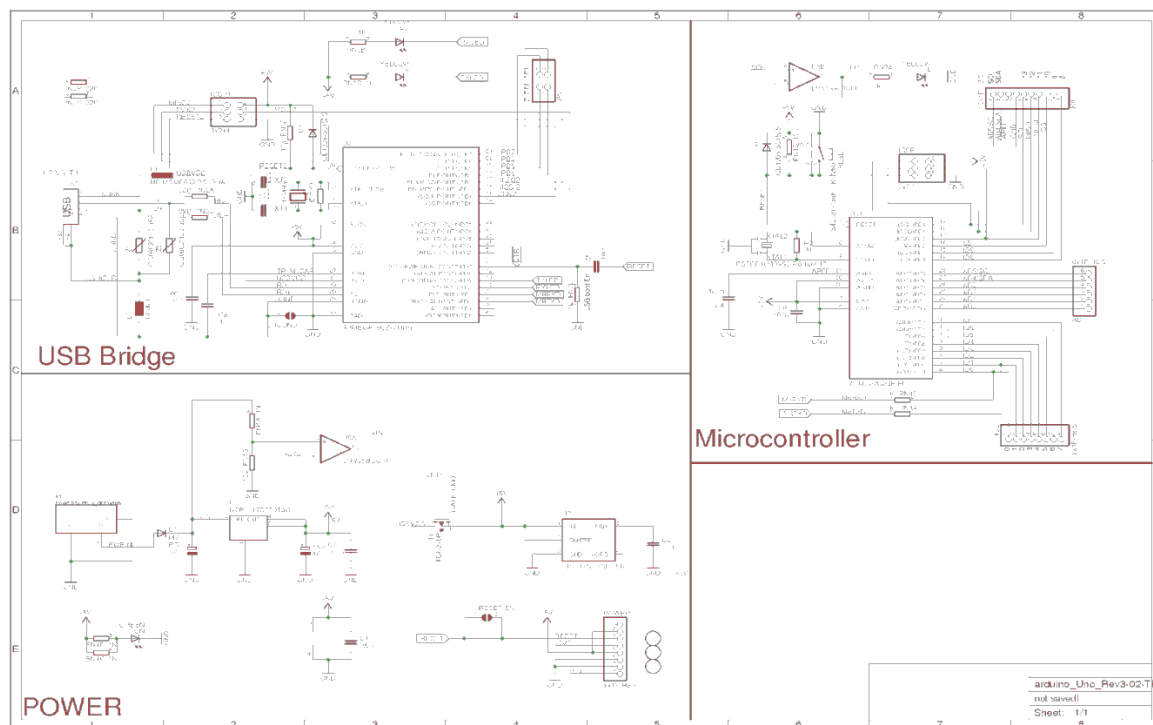
The dimensions of many SMD resistors, capacitors, and LEDs are indicated by package codes such as the following:

The bridge in the latest revision is the ATmega16U2, which has a USB transceiver and also a serial interface (UART interface).

To power your Arduino board, you can use the USB as a power source. Another option is to use a DC jack. You may ask, “if I connect both a DC adapter and the USB, which will be the power source?” The answer will be discussed in the “Power Part” section from this article.

To reset your board, you should use a push button in the board. Another source of reset should be every time you open the serial monitor from Arduino IDE.

I redistributed the original Arduino UNO schematic to be more readable below. I advise you to download it and open the PCB and schematic using Eagle CAD while you are reading this article.



The Microcontroller

It is important to understand that the Arduino board includes a microcontroller, and this microcontroller is what executes the instructions in your program. If you know this, you won't use the common nonsense phrase "Arduino is a microcontroller" ever again.

The ATmega328 microcontroller is the MCU used in Arduino UNO R3 as a main controller. ATmega328 is an MCU from the AVR family; it is an 8-bit device, which means that its data-bus architecture and internal registers are designed to handle 8 parallel data signals.

ATmega328 has three types of memory:

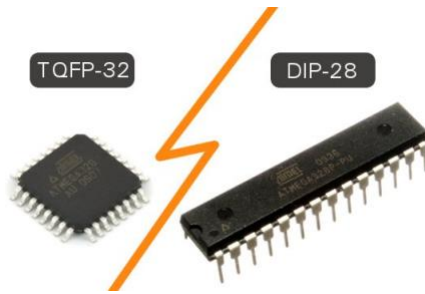
- **Flash memory:** 32KB nonvolatile memory. This is used for storing application, which explains why you don't need to upload your application every time you unplug arduino from its power source.
- **SRAM memory:** 2KB volatile memory. This is used for storing variables used by the application while it's running.
- **EEPROM memory:** 1KB nonvolatile memory. This can be used to store data that must be available even after the board is powered down and then powered up again.

Let us briefly go over some of this MCU's specs:

Packages:

This MCU is a DIP-28 package, which means that it has 28 pins in the dual in-line package. These pins include power and I/O pins. Most of the pins are multifunctional, which means that the same pin can be used in different modes based on how you configure it in the software. This reduces the necessary pin count, because the microcontroller does not require a separate pin for every function. It can also make your design more flexible, because one I/O connection can provide multiple types of functionality.

Other packages of ATmega328 are available like TQFP-32 SMD package (Surface Mount Device).



Two different packages of the ATmega328. Images courtesy of [Sparkfun](#) and [Wikimedia](#).

Power:

The MCU accepts supply voltages from 1.8 to 5.5 V. However, there are restrictions on the operating frequency; for example, if you want to use the maximum clock frequency (20 MHz), you need a supply voltage of at least 4.5 V.

Digital I/O:

This MCU has three ports: PORTC, PORTB, and PORTD. All pins of these ports can be used for general-purpose digital I/O or for the alternate functions indicated in the pinout below. For example, PORTC pin0 to pin5 can be ADC inputs instead of digital I/O.

There are also some pins that can be configured as PWM output. These pins are marked with “~” on the Arduino board.

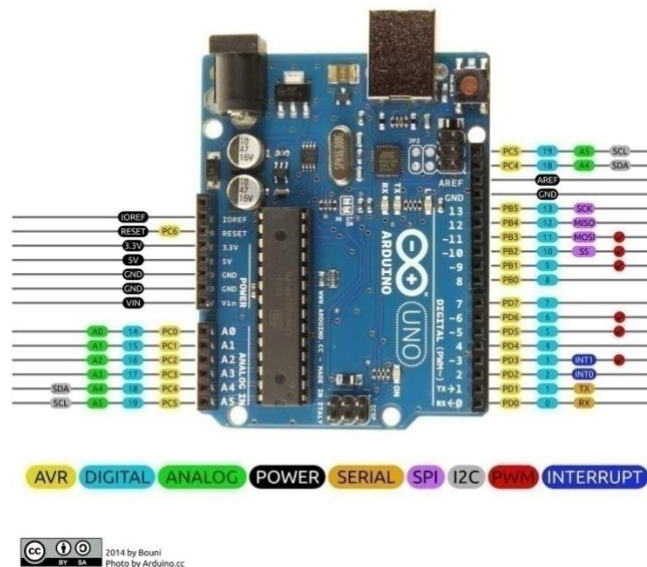
Note: The ATmega168 is almost identical to the ATmega328 and they are pin compatible. The difference is that the ATmega328 has more memory—32KB flash, 1KB EEPROM, and 2KB RAM compared to the ATmega168's 16KB flash, 512 bytes EEPROM, and 1KB RAM.

Atmega168 Pin Mapping			
Arduino function			Arduino function
reset	(PCINT14/RESET) PC6	1	PC5 (ADC5/SCL/PCINT13) analog input 5
digital pin 0 (RX)	(PCINT16/RXD) PD0	2	PC4 (ADC4/SDA/PCINT12) analog input 4
digital pin 1 (TX)	(PCINT17/TXD) PD1	3	PC3 (ADC3/PCINT11) analog input 3
digital pin 2	(PCINT18/INT0) PD2	4	PC2 (ADC2/PCINT10) analog input 2
digital pin 3 (PWM)	(PCINT19/OC2B/INT1) PD3	5	PC1 (ADC1/PCINT9) analog input 1
digital pin 4	(PCINT20/XCK/T0) PD4	6	PC0 (ADC0/PCINT8) analog input 0
VCC	VCC	7	GND
GND	GND	8	AREF analog reference
crystal	(PCINT6/XTAL1/TOSC1) PB6	9	AVCC VCC
crystal	(PCINT7/XTAL2/TOSC2) PB7	10	PB5 (SCK/PCINT5) digital pin 13
digital pin 5 (PWM)	(PCINT21/OC0B/T1) PD5	11	PB4 (MISO/PCINT4) digital pin 12
digital pin 6 (PWM)	(PCINT22/OC0A/AIN0) PD6	12	PB3 (MOSI/OC2A/PCINT3) digital pin 11 (PWM)
digital pin 7	(PCINT23/AIN1) PD7	13	PB2 (SS/OC1B/PCINT2) digital pin 10 (PWM)
digital pin 8	(PCINT0/CLKO/ICP1) PB0	14	PB1 (OC1A/PCINT1) digital pin 9 (PWM)

Digital Pins 11, 12 & 13 are used by the ICSP header for MOSI, MISO, SCK connections (Atmega168 pins 17, 18 & 19). Avoid low-impedance loads on these pins when using the ICSP header.

Atmega168 pinout with Arduino labels; the ATmega168 and ATmega328 are pin compatible. Image courtesy of [Arduino](#).

Arduino Uno R3 Pinout

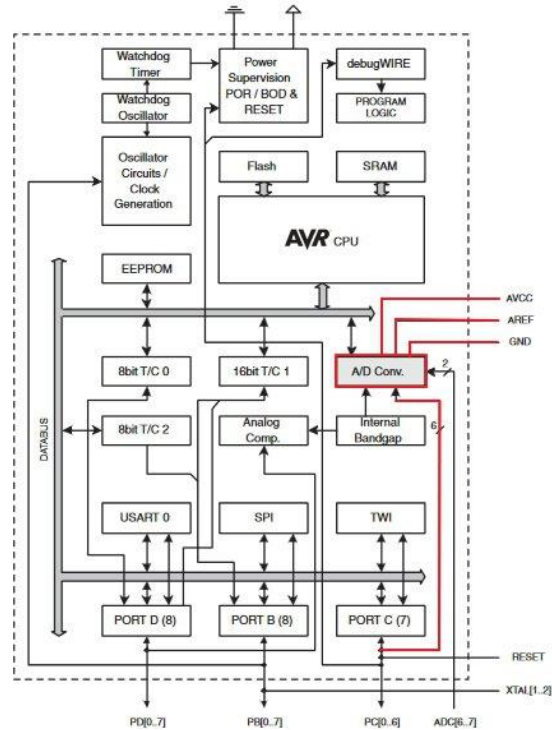


Arduino UNO R3 pinout. Image courtesy of [GitHub](#).

ADC Inputs:

This MCU has six channels—PORTC0 to PORTC5—with 10-bit resolution A/D converter. These pins are connected to the analog header on the Arduino board.

One common mistake is to think of analog input as dedicated input for A/D function only, as the header in the board states "Analog". The reality is that you can use them as digital I/O or A/D.



ATmega328 block diagram.

As shown in the diagram above (via the red traces), the pins related to the A/D unit are:

- AVCC: The power pin for the A/D unit.
- AREF: The input pin used optionally if you want to use an external voltage reference for ADC rather than the internal V_{ref}. You can configure that using an internal register.

Table 24-3. Voltage Reference Selections for ADC

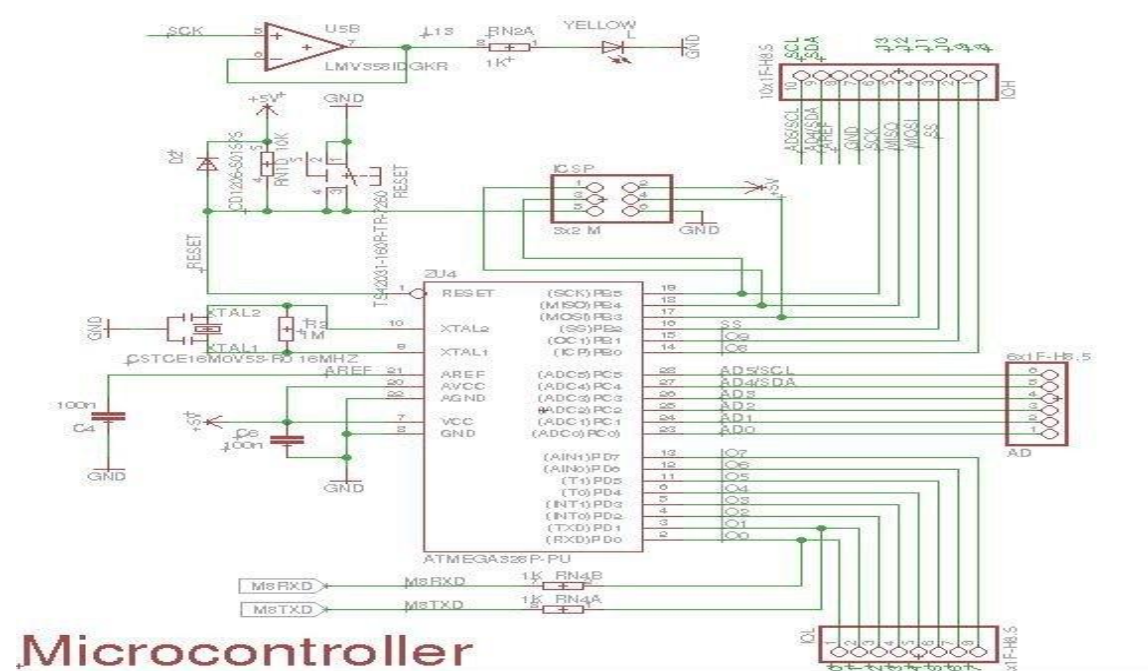
REFS1	REFS0	Voltage Reference Selection
0	0	AREF, Internal V _{ref} turned off
0	1	AV _{CC} with external capacitor at AREF pin
1	0	Reserved
1	1	Internal 1.1V Voltage Reference with external capacitor at AREF pin

Internal register settings for selecting the Vref source.

UART Peripheral:

A UART (Universal Asynchronous Receiver/Transmitter) is a serial interface. The ATmega328 has only one UART module.

The pins (RX, TX) of the UART are connected to a USB-to-UART converter circuit and also connected to pin0 and pin1 in the digital header. You must avoid using the UART if you're already using it to send/receive data over USB.



Arduino UNO R3 MCU part.

Returning to the electronic design, the microcontroller section has the following:

- **ATmega328-PU:** The MCU we just talked about.
- **IOL and IOH (Digital) Headers:** These headers are the digital header for pins 0 to 13 in addition to GND, AREF, SDA, and SCL. Note that RX and TX from the USB bridge are connected with pin0 and pin1.
- **AD Header:** The analog pins header.
- **16 MHz Ceramic Resonator (CSTCE16M0V53-R0):** Connected with XTAL2 and XTAL1 from the MCU.
- **Reset Pin:** This is pulled up with a 10K resistor to help prevent spurious resets in noisy environments; the pin has an internal pull-up resistor, but according to the AVR Hardware Design Considerations application note ([AVR042](#)), “if the environment is noisy, it can be insufficient and reset may occur sporadically.” Reset occurs if the user presses the reset button or if a reset is issued from the USB bridge. You can also see the D2 diode. The role of this diode is described in the same app note: “If not using High Voltage Programming it is recommended to add an ESD protection diode from RESET to Vcc, since this is not internally provided due to High Voltage Programming”.
- **C4 and C6 100nF Capacitors:** These are added to filter supply noise. The impedance of a capacitor decreases with frequency:

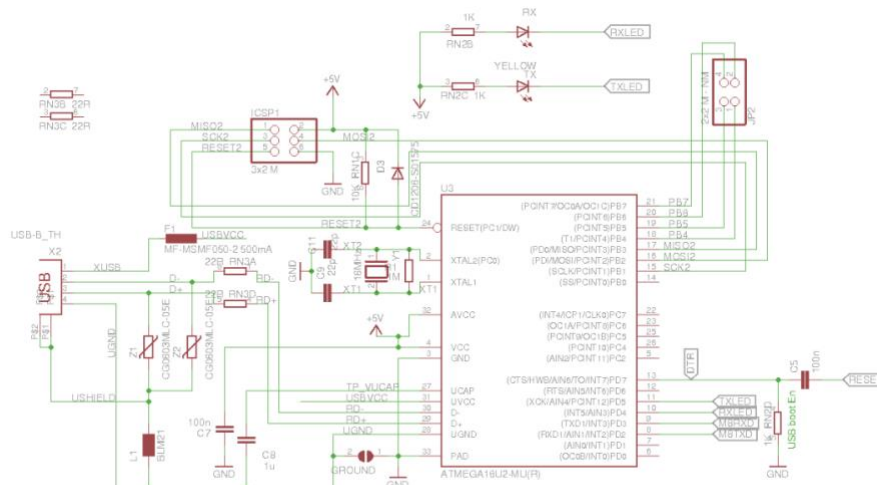
$$X_C = \frac{1}{2\pi f C}$$

The capacitors give high-frequency noise signals a low-impedance path to ground. 100nF is the most common value. Read more about capacitors in the [AAC textbook](#).

- **PIN13:** This is connected to the SCK pin from the MCU and is also connected to an LED. The Arduino board uses a buffer (the LMV358) to drive the LED.
- **ICSP (In-Circuit Serial Programming) Header:** This is used to program the ATmega328 using an external programmer. It's connected to the In-System Programming

(ISP) interface (which uses the SPI pins). Usually, you don't need to use this way of programming because bootloader handles the programming of the MCU from the UART interface which is connected using a bridge to the USB. This header is used when you need to flash the MCU, for example, with a bootloader for the first time in production.

The USB-to-UART Bridge



USB Bridge

Arduino USB bridge part.

As we discussed in the “Arduino UNO System Overview” section, the role of the USB-to-UART bridge part is to convert the signals of USB interface to the UART interface, which the ATmega328 understands, using an ATmega16U2 with an internal USB transceiver. This is done using special firmware uploaded to the ATmega16U2.

From an electronic design perspective, this section is similar to microcontroller section. This MCU has an ICSP header, an external crystal with load capacitors (CL), and a Vcc filter capacitor.

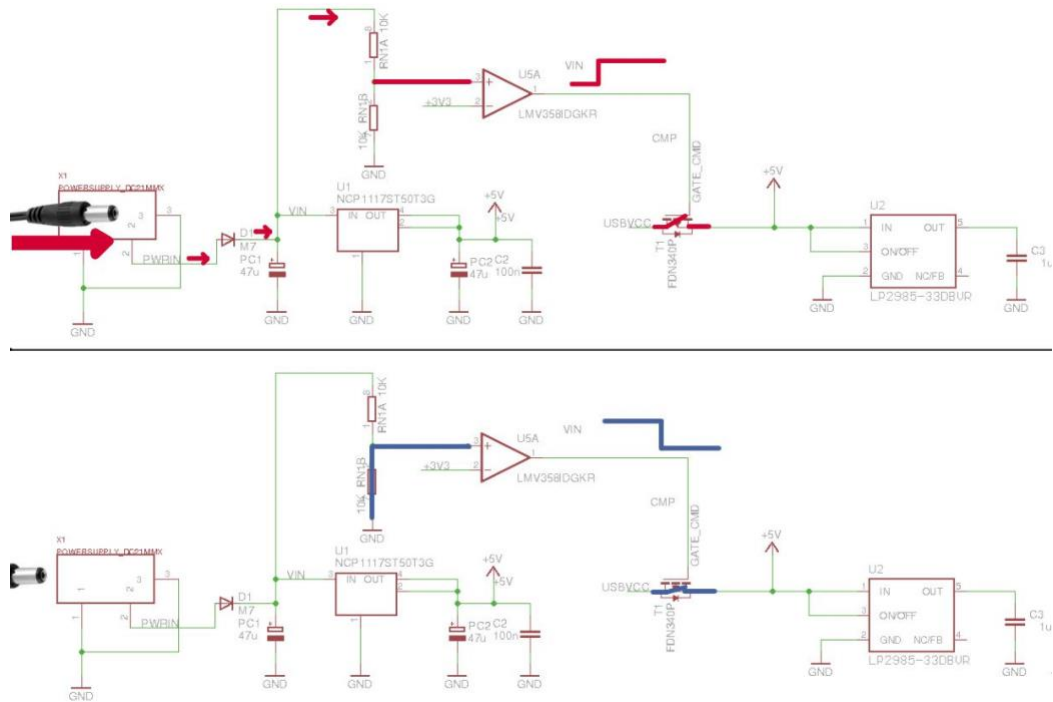
Notice that there are series resistors in the D+ and D- USB lines. These provide the proper termination impedance for the USB signals. Here is some further reading about these resistors:

1. [Why USB data series resistors](#)
2. [USB Developers FAQ](#)

Z1 and Z2 are voltage-dependent resistors (VDRs), also called varistors. They are used to protect the USB lines against ESD transients. The 100nF capacitor connected in series with the reset line allows the Atmega16U2 to send a reset pulse to the Atmega328. You can read more about this capacitor [here](#).

The PowerFor a power source, you have the option of using the USB or a DC jack. Now it's time to answer the following question: "If I connect both a DC adapter and the USB, which will be the power source?" The 5V regulator is the NCP1117ST50T3G and the Vin of this regulator is connected via DC jack input through the M7 diode, the SMD version of the famous [1N4007 diode](#) (PDF). This diode provides reverse-polarity protection.

The output of the 5V regulator is connected to the rest of 5V net in the circuit and also to the input of the 3.3V regulator, LP2985-33DBVR. You can access 5V directly from the power header 5V pin. Another source of 5V is USBVCC which is connected to the drain of an FDN340P, a P-channel MOSFET, and the source is connected to the 5V net. The gate of the transistor is connected to the output of an LMV358 op-amp used as a comparator. The comparison is between 3V3 and $V_{in}/2$. When $V_{in}/2$ is larger, this will produce a high output from the comparator and the P-channel MOSFET is off. If there is no Vin applied, the V+ of the comparator is pulled down to GND and Vout is low, such that the transistor is on and the USBVCC is connected to 5V.

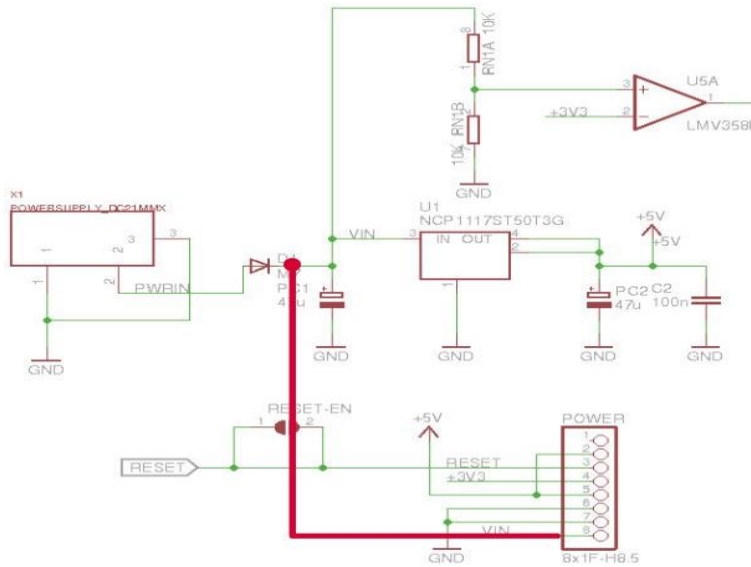


Power source switching mechanism.

The LP2985-33DBVR is the 3V3 regulator. Both the 3V3 and 5V regulators are LDO (Low Dropout), which means that they can regulate voltage even if the input voltage is close to the output voltage. This is an improvement over older linear regulators, such as the 7805.

The last thing I'll talk about is the power protection that is provided in Arduino UNO.

As mentioned above, VIN from a DC jack is protected from reverse polarity by using a serial M7 diode in the input. Be aware that the VIN pin in the power header is not protected. This is because it is connected after the M7 diode. Personally, I don't know why they decided to do that when they could connect it before the diode to provide the same protection.



When you use USB as a power source, and to provide protection for your USB port, there is a PTC (positive temperature coefficient) fuse (MF-MSMF050-2) in series with the USBVCC. This provides protection from overcurrent, 500mA. When an overcurrent limit is reached, the PTC resistance increases a lot. Resistance decreases after the overcurrent is removed.

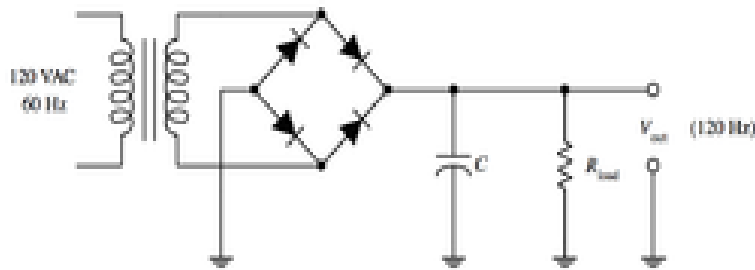
CHAPTER 4

DESCRIPTION OF PROJECT

7.1 DC Power Supply:

A DC power supply is one that supplies a voltage of fixed polarity (either positive or negative) to its load. Depending on its design, a DC power supply may be powered from a DC source or from an AC source such as the power mains.

AC-to-DC supply



Schematic of basic AC-to-DC power supply, showing (from L-R) transformer, full-wave bridge rectifier, filter capacitor and resistor load

Some DC power supplies use AC mains electricity as an energy source. Such power supplies will sometimes employ a transformer to convert the input voltage to a higher or lower AC voltage. A rectifier is used to convert the transformer output voltage to a varying DC voltage, which in turn is passed through an electronic filter to convert it to an unregulated DC voltage. The filter removes most, but not all of the AC voltage variations; the remaining voltage variations are known as *ripple*.

The electric load's tolerance of ripple dictates the minimum amount of filtering that must be provided by a power supply. In some applications, high ripple is tolerated and therefore no filtering is required. For example, in some battery charging applications it is possible to implement a mains-powered DC power supply with nothing more than a transformer and a single rectifier diode, with a resistor in series with the output to limit charging current.

Overview

This article covers the design of basic unregulated DC power supplies. It is not designed to replace the training given by any college or university, but you may find it a useful study supplement. **Load Requirements**

Before designing any power supply the load requirements must be known. It is always a good idea to take the worst case scenario when making this decision. For example if your circuit is designed to draw 1 amp at 12 volts, assume that component tolerances are 20% and design to meet these requirements with at least 20-50% reserve current, in this example I would design a power supply which could safely deliver 12 volts at 1.5 amps without overheating.

Transformer Regulation and Efficiency
A transformer is very efficient at converting AC voltages and currents from one value to another. In practice efficiencies of 98% may be achieved, the losses being due to heating effects of the transformer core, winding loss and leakage flux.

Transformers have VA ratings which is simply the secondary voltage multiplied by secondary current -- this is strictly true only if the attached load is purely resistive (i.e has a power factor of 1.0). A reactive load containing capacitors or inductors (which one would expect for such a power supply) has a low power factor (i.e. less than 1.0) and thus de-rates the transformer's power capacity to the stated VA multiplied by the power factor because it draws more current than a purely resistive load. So, when choosing a transformer for a reactive load, one needs to divide the load in watts by the load's power factor to arrive at the VA needed which has sufficient "headroom" to accommodate the low power factor.

Not often published are the regulation figures for a typical transformer. A transformer rated at 20 V , 1 A secondary will only measure 20 volts when it is actually delivering 1 A. The figures below show typical regulation figures for some common VA rated transformers:-

VA Rating	6	12	20	50	100
% Regulation	25	12	10	10	10

For example a 12 VA rated transformer would have a no-load voltage which is 12% higher than the rated value. If the transformer was rated at 12 V @ 1 A, when measuring the secondary RMS voltage with a high impedance meter, you would measure approximately 13.44 Volts.

Rectification - Unfiltered Power Supplies

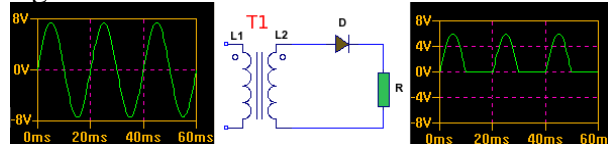
This is the process where alternating current is converted to direct current. Unfiltered, means that there is no smoothing capacitor present and the dc output will contain "ripples" at the line (mains) frequency. There are two types of rectification, half wave and full wave, also known as a bridge rectifier.

Half-Wave

The half wave rectifier circuit is shown in Fig. 2 below:

Figure

2



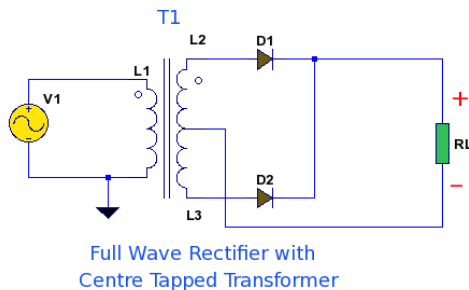
The DC output across a resistive load, is approximately the value of a half cycle, less one diode drop. Rectifier diodes have a forward voltage that varies from about 0.7V to 1.1 Volts in high current rectifiers. Conduction occurs for only one cycle, so is not very efficient, also without a smoothing capacitor, the output is quite "lumpy". Often these are used in cheap car battery chargers where the quality of the supply is not too important.

Full-Wave using Centre Tapped Transformer

A full wave rectifier circuit using a centre tapped transformer is shown below in Figure 3. This circuit uses just two diodes each one conducting on alternate half cycles. The positive side is marked with a "+" and the output waveform shown in figure 5. Notice that the output ripple is now doubled.

Figure

3

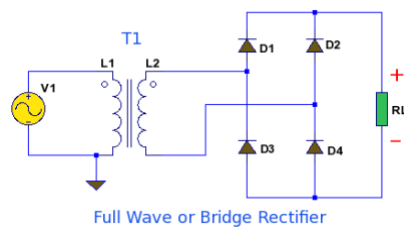


Full-Wave or Bridge Rectifier

The bridge rectifier is the most popular rectifier circuit. It uses four diodes arranged in a ring, but complete four terminal bridge rectifiers are also available. The circuit is shown in figure 4 below:

Figure

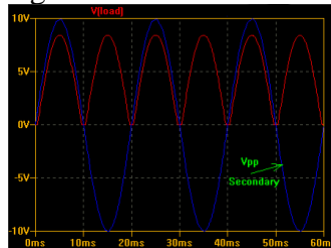
4



There are twice the amount of "peaks" compared to the half wave rectifier because alternate diode pairs conduct for each half cycle of the AC input. A typical waveform is shown below (Figure 5):

Figure

5



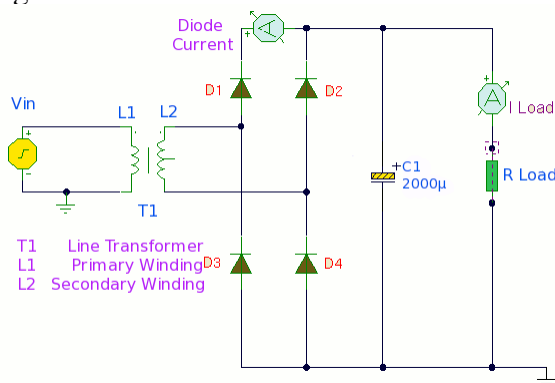
The blue trace is the peak to peak voltage of the transformers secondary winding, and the red trace is the unfiltered DC voltage. The DC output is approximately: $1.41 \times V_{RMS} - (2 \times 0.7)$

Rectification - Filtered Power Supplies

The "raw" DC produced after rectification is OK to charge a battery or light a lamp but any electronic circuit needs a smooth DC supply. In the case of audio circuits, particularly amplifiers, any unfiltered DC will be heard as a "hum" in the equipment's loudspeakers. The hum is proportional to the AC power supply's frequency. A filtered or smoothed supply is achieved by placing a large value electrolytic capacitor at the rectifiers output, as shown in Fig. 6 below.

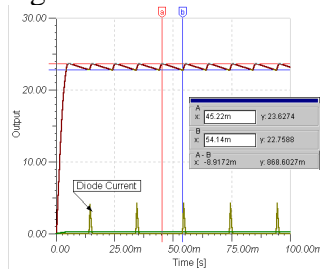
Figure

6



The resulting waveforms are drawn in Fig. 7 below. The "brown" waveform represents the filtered DC feeding the load resistor.

Figure



Ripple Voltage

The rectifier diodes will charge up the filter capacitor, C1 to the peak DC value, and between non conducting cycles of the diodes, will discharge into the load resistor. This creates the sawtooth waveform known more commonly as ripple voltage. The value of the ripple voltage is dependent on load current, power supply frequency and capacitor value. Approximate ripple voltage is calculated using:

$$\Delta V = \frac{I}{2fC}$$

where V is ripple voltage (mV), I is DC load current (mA), f is frequency of AC supply and C is smoothing capacitor value (F). For countries using a 50Hz supply like the United Kingdom, then the following simplified equation will also give the same results:

$$\Delta V = \frac{10I}{C}$$

where V is ripple voltage (V), I is load current (mA) and C is smoothing capacitor (uF).

Example: The bridge rectifier circuit above had a load current of about 191mA. Feeding this value into the first equation results in $191/(2 \times 50 \times 2200 \times 10^{-6}) = 868.1 \text{ mV}$ and the bottom equation $(10 \times 191)/2200 = 0.868\text{V}$ or 868mV

As where:

RMS

T_1 = Diode conduction time

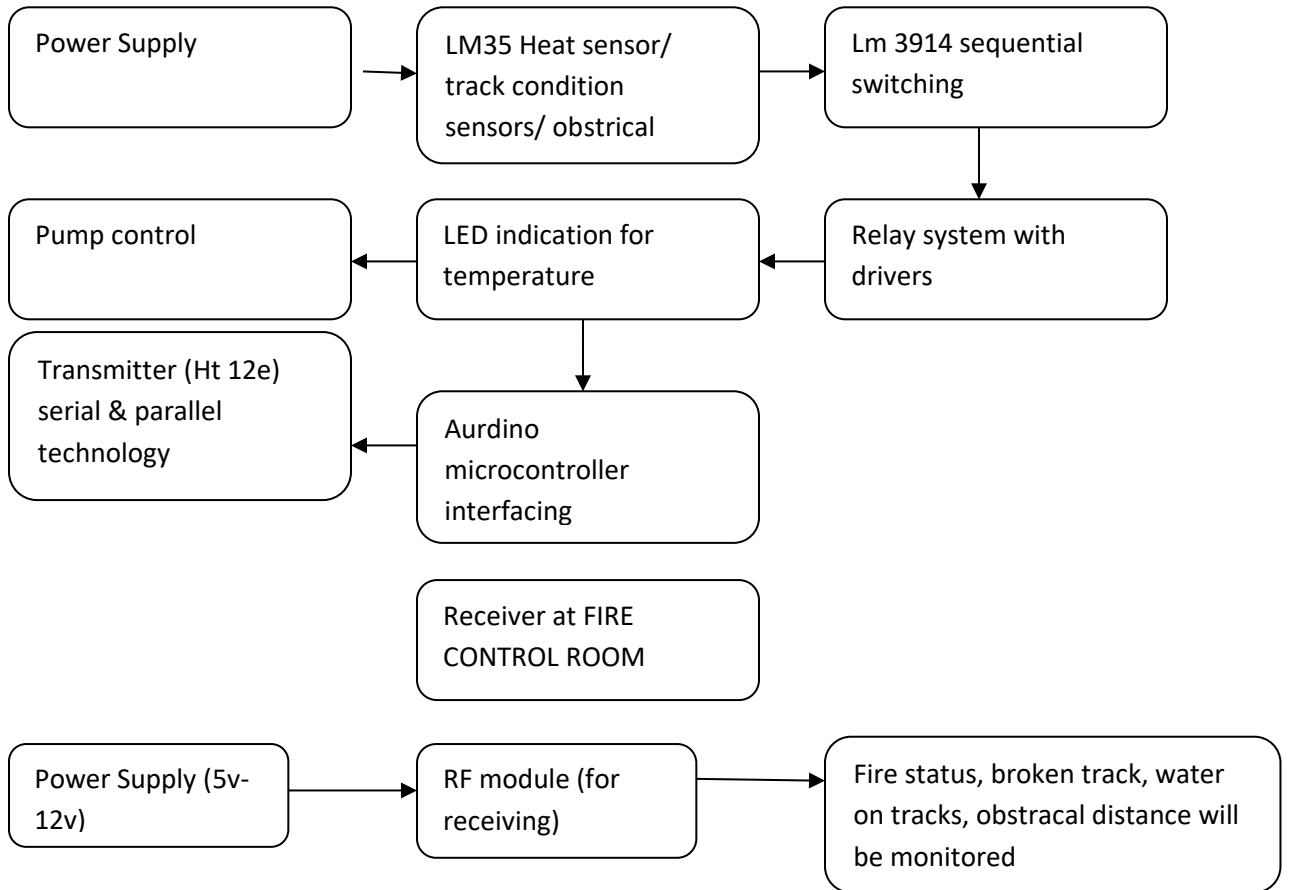
T = $1/f$ f being line frequency

I_{dc} = Average load current

I_{peak} = Peak current through the rectifier

Fire Monitoring

Block Diagram



RF Module



The RF module, operates at Radio Frequency. The corresponding frequency range varies between 30 kHz & 300 GHz. In this RF system, the digital data is represented as variations in the amplitude of carrier wave. This kind of modulation is known as Amplitude Shift Keying (ASK).

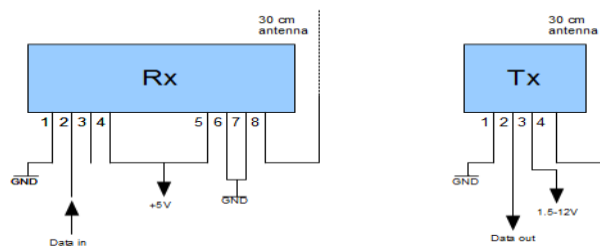
Why RF ?

Transmission through RF is better than IR (infrared) because of many reasons. Firstly, signals through RF can travel through larger distances making it suitable for long range applications. Also, while IR mostly operates in line-of-sight mode, RF signals can travel even when there is an obstruction between transmitter & receiver. Next, RF transmission is more strong and reliable than IR transmission. RF communication uses a specific frequency unlike IR signals which are affected by other IR emitting sources.

This **RF module** comprises of an **RF Transmitter** and an **RF Receiver**. The transmitter/receiver (Tx/Rx) pair operates at a frequency of **434 MHz**. An RF transmitter receives serial data and transmits it wirelessly through RF through its antenna connected at pin4. The transmission occurs at the rate of 1Kbps – 10Kbps. The transmitted data is received by an RF receiver operating at the same frequency as that of the transmitter.

The RF module is often used alongwith a pair of encoder/decoder. The encoder is used for encoding parallel data for transmission feed while reception is decoded by a decoder. HT12E-HT12D, HT640-HT648, etc. are some commonly used encoder/decoder pair ICs.

PIN DIAGRAM



RF Communication Between Microcontrollers – Part II

- Posted On 11 Aug 2009 Welcome back to the Part II of [RF Communication tutorial](#). Here I will show you the basic working of RF modules and how to send and receive data. Please see the [Part I](#) of this tutorial for basic introduction. You should also be familiar with RS232 communication. If you are new to it please see [RS232 Serial Communication Tutorial](#). I also recommend using wireless link only after you have successfully tried wired RS232 communication. Here I will not go deep in how RS232 works because it is already discussed in [RS232 Serial Communication Tutorial](#). I will use my interrupt driven fully buffered USART library for communication.

How RF Module Works

Working of RF Modules is simple but with a little trick. The working is shown in figure below.

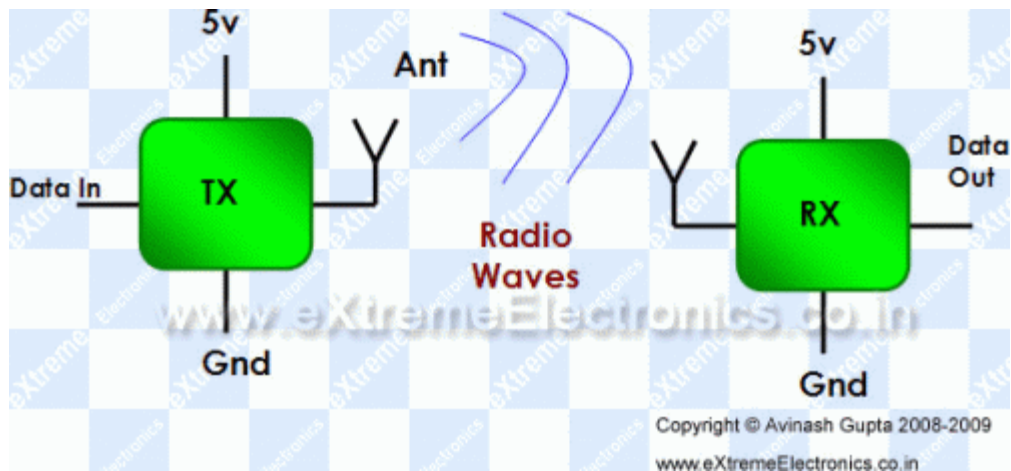


Fig- Working of RF Modules.

Here what ever digital data you input on "**Data In**" of TX is available on "**Data Out**" of RX. Say, if you set "data in" high, the "data out" will become high as well. But here lies the trick! The fact is that you cannot Keep Logic HIGH or LOW for a Long period of time, say for a few millisecond second. If you apply a logic low on "data in" the "data out" will become low but only for few millisecond and it will start oscillating(become high/low repeatedly) after that. Same thing will happen if you set "data in" to high state.

Let us assume we have connected the RF modules with MCUs as shown below.

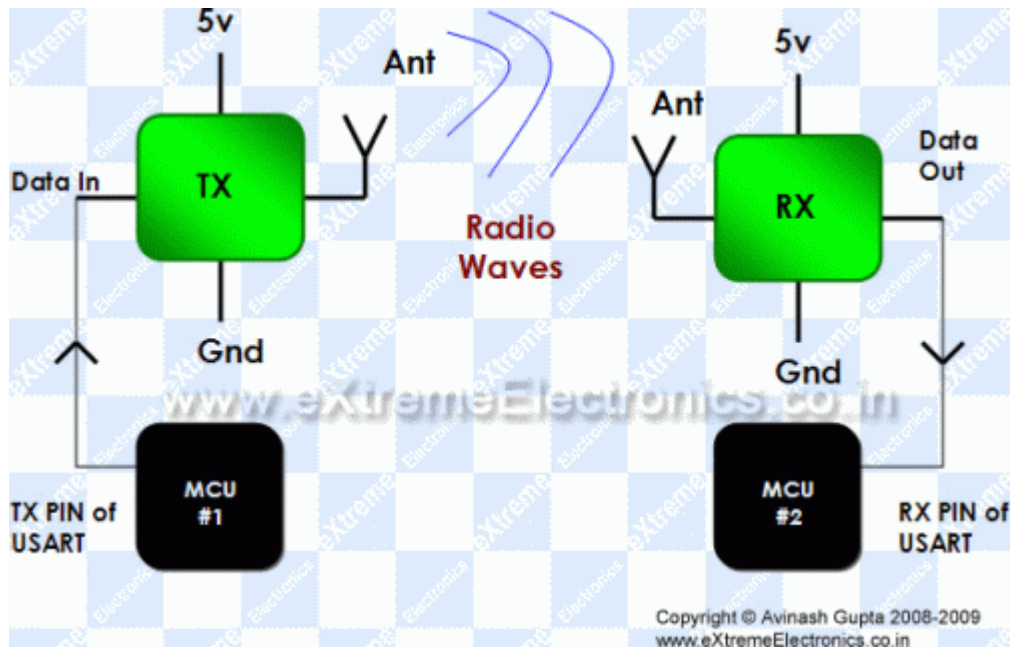


Fig- RF Modules connected to USART of Microcontrollers.

When the TX unit is switched off or not transmitting data, then as I said the "**data out**" of RX will be oscillating high and low and as this is connected to RX of MCU's USART, the MCU#2 will be receiving garbage data. And when TX unit will send some data, MCU#2 will also be receiving them. So MCU #2 is always receiving data, even when MCU#1 is not sending anything. So there must be a mechanism to differentiate real data with garbage data.

Sending and receiving data.

For this I created a simple mechanism. The steps are shown below.

- We begin transmission by sending character 'A'
- We again send one more 'A'
- Then we send the actual data.
- Now we send the **inverse** of data. That is all 0's are converted to 1 and vice-versa.
- We end the packet by sending 'Z'

In this way we create a simple packet based transmission with error detection. Now in the RX side MCU our program follows the algorithm given below.

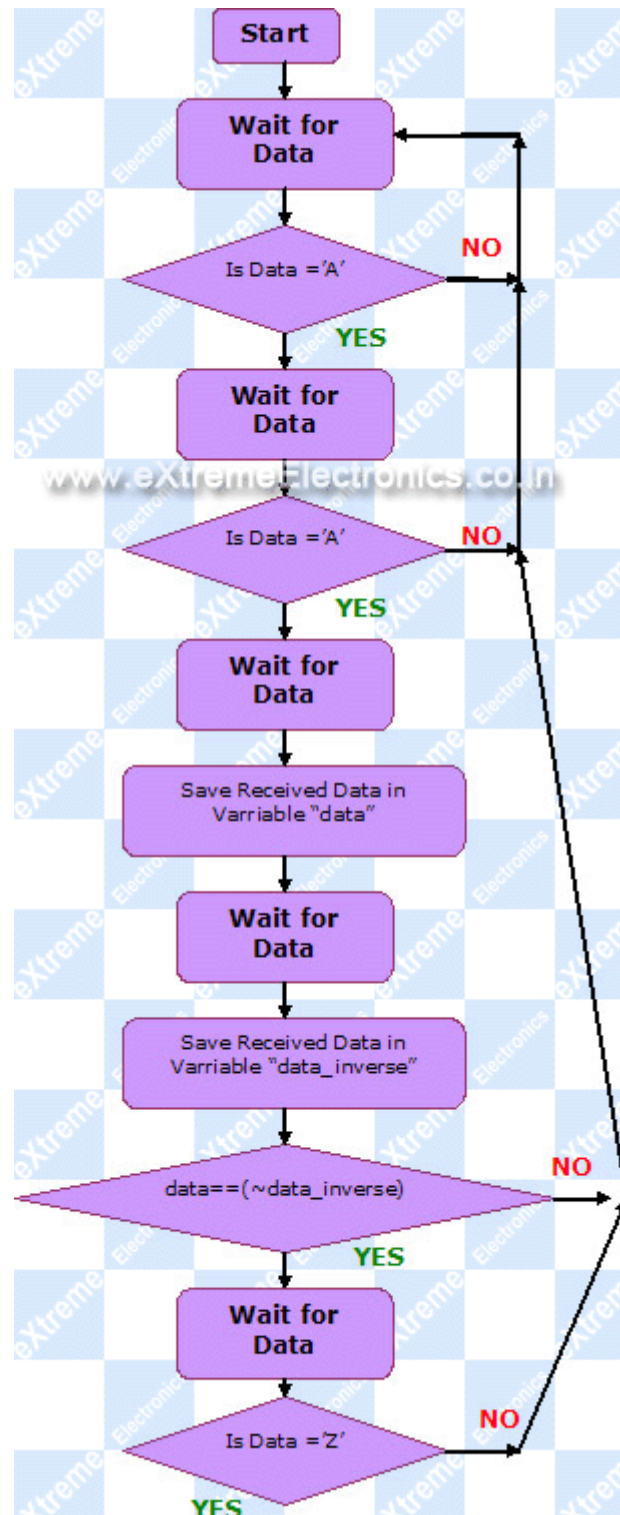


Fig- Data Reception Algorithm.

If we went to the end of algorithm successfully it means that we have got a valid data and we can use it.

So in this way we saw how a byte of data is transmitted from MCU#1 to MCU#2 via air. This simple algorithm filters real data from garbage data. In the next tutorial we will see the practical example of data transmission and reception. We will use the received data to control the IO ports of MCU, in this way we will create a simple multi channel wireless remote control.

[Go to Part III](#)

By

[Avinash Gupta](#)

me@avinashgupta.com

[My Facebook Profile](#)



RF Modules


We highly recommend **EasyEDA** for Circuit Design and **low cost PCB Prototype**.

Try this tool for an Easy-to-Use way to design circuits and layout PCBs

Only \$9.8 for 10 pcs Circuit Boards, quick delivery,100% E-test

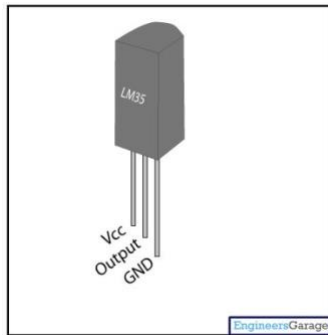
LM 35 Fire Sensor

- Use a conversion factor that is the reciprocal, that is 100 °C/V.
- The general equation used to convert output voltage to temperature is:
 - Temperature (°C) = Vout * (100 °C/V)
 - So if Vout is 1V , then, Temperature = 100 °C
 - The output voltage varies linearly with temperature.



The operating temperature range is from -55°C to 150°C . The output voltage varies by 10mV in response to every $^{\circ}\text{C}$ rise/fall in ambient temperature, *i.e.*, its scale factor is $0.01\text{V}/^{\circ}\text{C}$.

Pin Diagram:



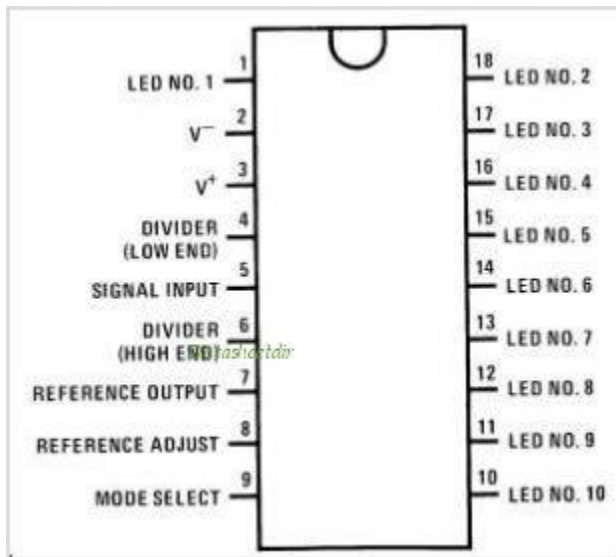
Pin Description:

Pin No	Function	Name
1	Supply voltage; 5V (+35V to -2V)	Vcc
2	Output voltage (+6V to -1V)	Output
3	Ground (0V)	Ground

Sequential switching IC LM 3914

LM3914 is a monolithic integrated circuit that senses analog voltage levels and drives 10 LEDs, providing a linear analog display. A single pin changes the display from a moving dot to a bar graph. Current drive to the LEDs is regulated and programmable, eliminating the need for resistors. This feature is one that allows operation of the whole system from less than 3V.

LM3914 datasheet



LM3914 features

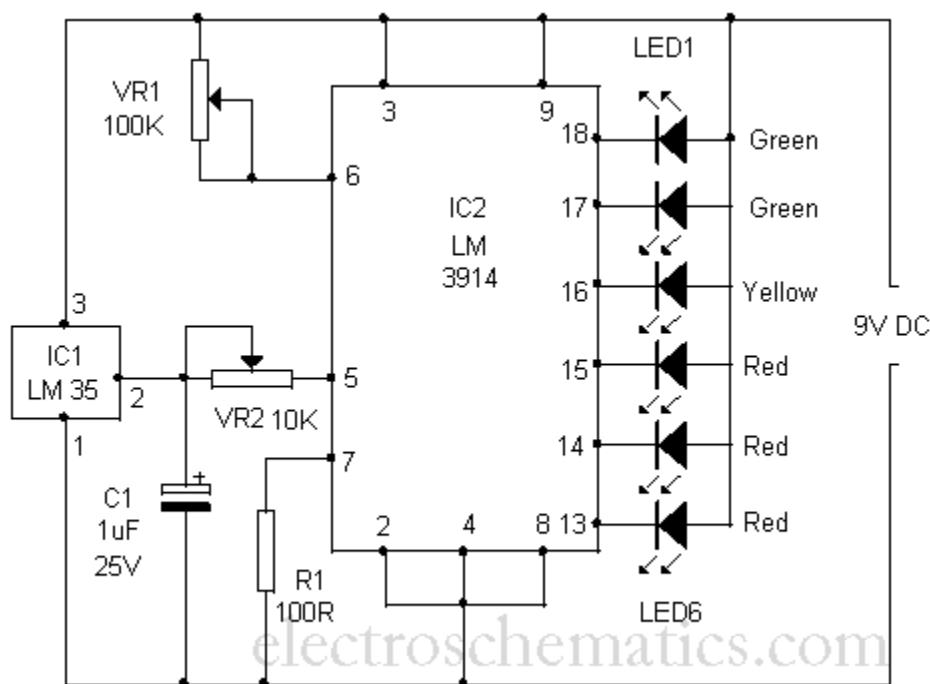
- Drives LEDs, LCDs or vacuum fluorescents
- Bar or dot display mode externally selectable by user
- Expandable to displays of 100 steps
- Internal voltage reference from 1.2V to 12V
- Operates with single supply of less than 3V
- Inputs operate down to ground
- Output current programmable from 2 mA to 30 mA
- No multiplex switching or interaction between outputs
- Input withstands 35V without damage or false outputs
- LED driver outputs are current regulated, open-collectors
- Outputs can interface with TTL or CMOS logic
- The internal 10-step divider is floating and can be referenced to a wide range of voltages

LM 35 and LM3914 Interfacing Circuit Diagram

This temperature meter uses the precision micro power centigrade sensor IC LM35. The output voltage of the IC is linearly equal to 10mV per degree centigrade. The temperature level is displayed through LED readout. The circuit uses the **precision temperature IC LM35**. This three pin transistor like IC give output linearly equal to **10mV** per degree rise in temperature. It can measure temperature between **-4 degree to 110 degree** centigrade. Its related type LM34 is Fahrenheit sensor and its output is equal to -10mV per degree Fahrenheit. Output of IC1 is directly given into the input of the display driver **IC LM3914**. It is a monolithic integrated circuit with 10 active low outputs that can drive 10 LEDs directly without a current limiting resistor.

The internal circuitry of the IC adjusts the current passing through the LEDs. The input of LM 3914 is very sensitive and its outputs 18 – 10 sinks current one by one as the input receives an increment of **125 milli volts**. Here only 6 outputs are used to drive 6 LEDs. More LEDs can be included in the remaining outputs if required. As the IC LM35 senses temperature rise, LEDs one to six light up. If the sensitivity is not high, VR2 can be omitted. Then output of IC1 should be directly connected to the input of IC2

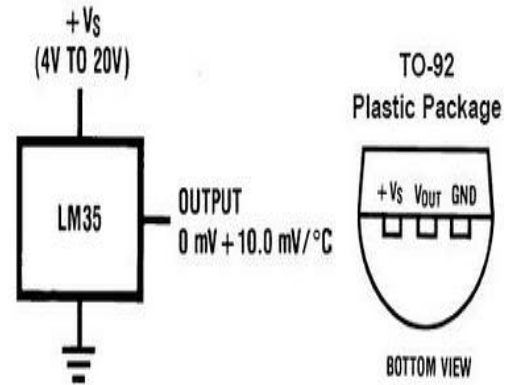
Temperature Meter Circuit diagram



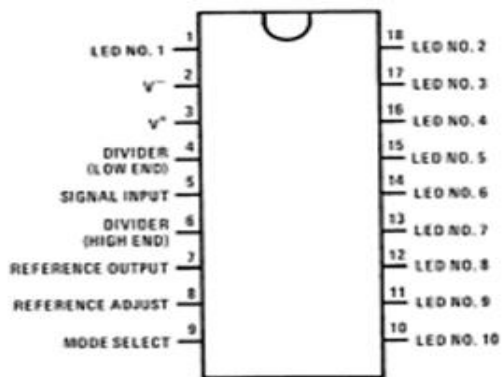
Calibration

When power is applied, some of the LEDs will glow. Calibrate the circuit by giving different temperature to IC1. For this a thermometer and hot water of different temperature is required. Soak some cotton with warm water of around 37degree (normal room temperature) and gently make contact with IC1. Adjust VR1 and VR2, till LED1 glows.

IC LM35 and LM3914



IC LM35 and LM 3914 Pin out



LM3914 datasheet

Relay with Relay Driver



A relay driver circuit is a circuit which can drive, or operate, a relay so that it can function appropriately in a circuit.

The driven relay can then operate as a switch in the circuit which can open or close, according to the needs of the circuit and its operation.

In this project, we will build a relay driver for both DC and AC relays. Since DC and AC voltages operate differently, to build relay drivers for them requires slightly different setup. We will also go over a generic relay driver which can operate from either AC or DC voltage and operate both AC and DC relays.

All the circuits are relatively simple to understand.

DC Relay Driver Circuit

We will first go over how to build a relay driver circuit for relays which operate from DC power.

To drive a DC relay, all we need is sufficient DC voltage which the relay is rated for and a zener diode.

All relays come with a voltage rating. This is called on a relay's datasheet its rated coil voltage. This is the voltage needed in order for the relay to be able to operate and be able to open or close its switch in a circuit. In order for a relay to function, it must receive this voltage at its coil terminals. Thus, if a relay has a rated voltage of 9VDC, it must receive 9 volts of DC voltage to operate. So the most important thing a DC relay needs is its rated DC voltage. If you don't know this, look up what relay you have and look up its datasheet and check for this specification.

And the reason why a diode is needed is usually because it functions to eliminate voltage spikes from a relay circuit as the relay opens and closes. The coil of a relay acts as an inductor. Remember that inductors are basically coils of wires wrapped around a conductive core. This is what relay coils are as well. Therefore, they act as inductors. Inductors are electronic components that resist changes in current. Inductors do not like sudden changes in current. If the flow of current

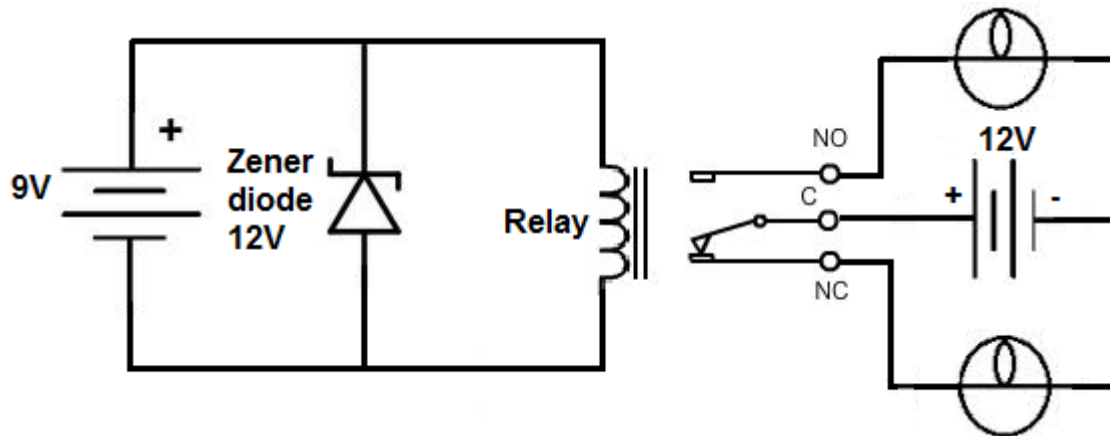
through a coil is suddenly interrupted, for example, a switch opening, the coil will respond by producing a sudden, very large voltage across its leads, causing a large surge of current through it. From a physics or physical perspective, this phenomenon is a result of a collapsing magnetic field within the coil as the current is terminated abruptly. Mathematically, this can be understood by noticing how a large change in current (dI/dt) affects the voltage across a coil ($V=LdI/dt$). Since we are opening the switch, in this case, the current literally goes from full mode to 0 instantaneously. This creates a large voltage spike. Surges in current that result from inductive effects can create very high voltage spikes (as high as 1000V) that can have nasty effects on neighboring devices within the circuits, such as switches and transistors getting zapped. Not only are these voltage spikes damaging to other electronic components in a circuit but they are also damaging to the relay's switch contacts. The contacts will suffer from these spikes as well.

So how do we prevent these voltage spikes? How can we suppress them so that they don't cause this damage? The answer for DC relay circuits is to use a diode. A diode is placed reverse biased in parallel with the relay. The diode acts as a transient suppressor. A transient is a spike. A transient suppressor suppresses these spikes. Placing a diode in reverse bias across a relay's coil eliminates voltage spikes by going into conduction before a large voltage can form across the coil. In other words, a diode will conduct current in reverse bias once the voltage reaches a certain threshold and shunt the current to ground. Once the diode begins conducting, it no longer holds voltage. So that the relay in parallel will not receive the excess voltage. So the diode functions to shunt excess power to ground once it reaches a certain threshold. Diodes are devices that do not conduct in reverse. However, if the voltage reaches a certain level, called the breakdown voltage, it will conduct. This is a good thing, when we need the diode to act as a transient suppressor, because it forces all excess power to ground, as to not affect any other parts of the circuit.

The diode must be rated to handle currents equivalent to the maximum current that would have been flowing through the coil before the supply current was interrupted. Therefore, if the relay normally passes a certain amount of current through it during normal operation, the diode must be rated for a current rating above this value, as to not stop normal operation.

DC Relay Driver Circuit Schematic

Below is the DC relay driver circuit which we will build:



The relay which we use in this case is rated for 9V. Therefore, a 9-volt DC voltage source feeds the resistor. To suppress transients that may be caused by the relay opening and closing, we place a zener diode reverse biased in parallel with the relay. This will shunt all excess power to ground once it reaches a certain threshold. This is all that is needed to operate the relay. With sufficient power, the relay will now closed, driving the loads that are connected to its output.

How to Build an AC Relay Driver Circuit

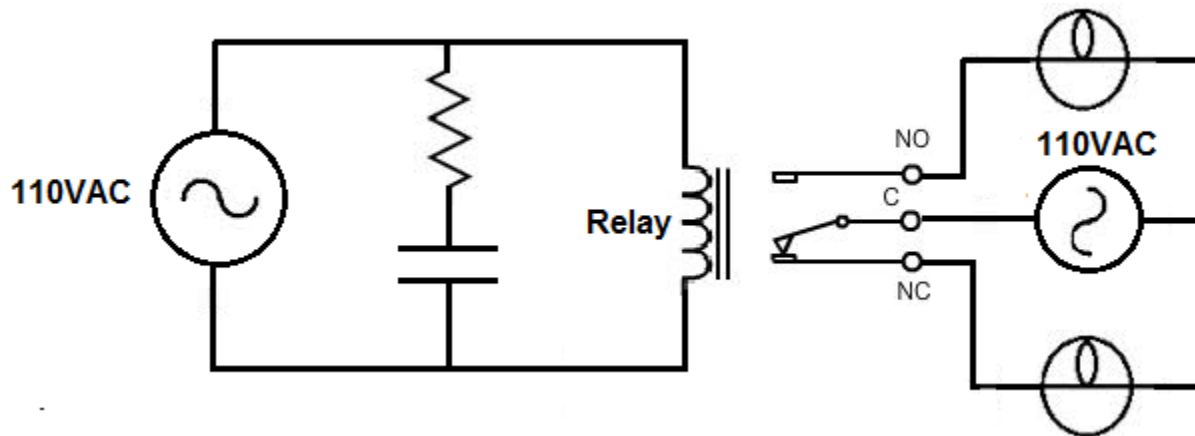
Now we will show how to build an AC relay driver circuit.

This is a relay which is run, not off of DC power, but AC power.

To drive an AC relay, all we need is sufficient AC voltage which the relay is rated for and again a transient suppressor.

Unlike DC relays, however, you cannot use a diode to to eliminate voltage spikes. With AC power, the diode will conduct on alternate half-cycles. Using 2 diodes in reverse parallel will also not work because the current will not make it to the coil of the relay. The current will just go through the diodes. Instead, to create a working transient voltage suppressor with an AC circuit, we use an RC series newtwork placed across the coil in parallel. The capacitor absorbs excessive charge and the resistor helps to control the discharge.

The AC relay driver circuit we will build is shown below:



We, again, feed the AC relay the AC voltage it is rated for. If we use a relay with a rated voltage of 110VAC, we must feed it 110V from an AC power source. The capacitor and resistor in series acts as the transient voltage suppressor to suppress voltage spikes. This first half of the circuit serves as the relay driver. With the relay now having sufficient power, it will turn on and power the loads it is connected to.

Generic Relay Driver Circuit

The last relay driver circuit we will show is one which can be driven by an arbitrary control voltage.

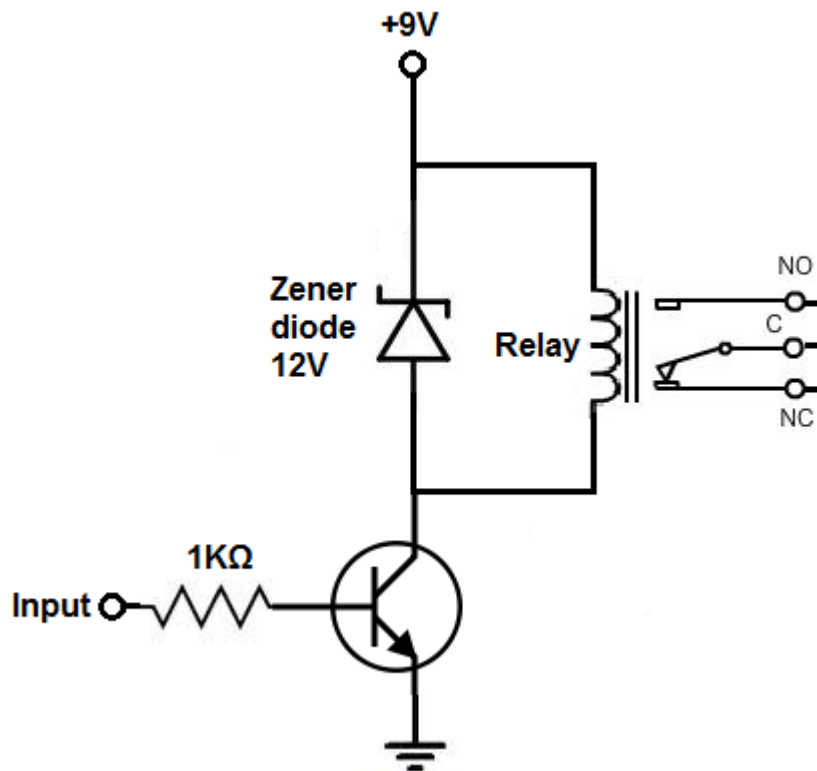
This is a relay driver circuit which can be driven by either AC or DC input voltage. And unlike the other circuits, a specific voltage, such as the rated voltage values we used to drive the others, does not need to be used. Because this circuit contains a transistor, much less power needs to be used on the input side to drive it.

Components Needed

- 6-9V Relay
- 2N2222 Transistor
- Zener diode
- $1K\Omega$ Resistor
- 9V Battery or DC Power Supply
- Another input voltage source

Relay Driver Circuit

The circuit we will build is shown below:



Now that we're using a transistor to drive the relay, we can use considerably less power to get the relay driven. Because a transistor is an amplifier, we just have to make sure that the base lead gets enough current to cause a larger current to flow from the emitter of the transistor to the collector. Once the base receives sufficient power, the transistor will conduct from emitter to collector and power the relay.

With no voltage or input current applied to the transistor's base lead, the transistor's emitter-to-collector channel is open, hence blocking current flow through the relay's coil. However, if sufficient voltage and input current are applied to the base lead, the transistor's emitter-to-collector channel will open, allowing current to flow through the relay's coil.

The benefit of this circuit is a smaller and arbitrary (DC or AC) current can be used to power the circuit and the relay.

Interfacing Temperature Sensing

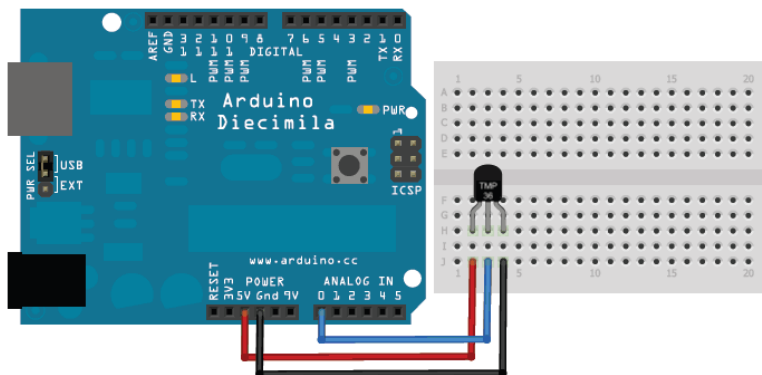
Connecting to a Temperature Sensor

These sensors have little chips in them and while they're not that delicate, they do need to be handled properly. Be careful of static electricity when handling them and make sure the power supply is connected up correctly and is between 2.7 and 5.5V DC - so don't try to use a 9V battery!

They come in a "TO-92" package which means the chip is housed in a plastic hemi-cylinder with three legs. The legs can be bent easily to allow the sensor to be plugged into a breadboard. You can also solder to the pins to connect long wires. If you need to waterproof the sensor, you can see below for an Instructable for how to make an excellent case.

Reading the Analog Temperature Data

Unlike the FSR or photocell sensors we have looked at, the TMP36 and friends doesn't act like a resistor. Because of that, there is really only one way to read the temperature value from the sensor, and that is plugging the output pin directly into an Analog (ADC) input.



Remember that you can use anywhere between 2.7V and 5.5V as the power supply. For this example I'm showing it with a 5V supply but note that you can use this with a 3.3v supply just as easily. No matter what supply you use, the analog voltage reading will range from about 0V (ground) to about 1.75V.

If you're using a 5V Arduino, and connecting the sensor directly into an Analog pin, you can use these formulas to turn the 10-bit analog reading into a temperature:

Voltage at pin in milliVolts = (reading from ADC) * (5000/1024)

This formula converts the number 0-1023 from the ADC into 0-5000mV (= 5V)

If you're using a 3.3V Arduino, you'll want to use this:

Voltage at pin in milliVolts = (*reading from ADC*) * (3300/1024)

This formula converts the number 0-1023 from the ADC into 0-3300mV (= 3.3V)

Then, to convert millivolts into temperature, use this formula:

Centigrade temperature = [(analog voltage in mV) - 500] / 10

Simple Thermometer

This example code for Arduino shows a quick way to create a temperature sensor, it simply prints to the serial port what the current temperature is in both Celsius and Fahrenheit.

```
//TMP36 Pin Variables
int sensorPin = 0; //the analog pin the TMP36's Vout (sense) pin is connected to
                    //the resolution is 10 mV / degree centigrade with a
                    //500 mV offset to allow for negative temperatures

/*
 * setup() - this function runs once when you turn your Arduino on
 * We initialize the serial connection with the computer
 */
void setup()
{
  Serial.begin(9600); //Start the serial connection with the computer
                    //to view the result open the serial monitor
}

void loop()          // run over and over again
{
  //getting the voltage reading from the temperature sensor
  int reading = analogRead(sensorPin);

  // converting that reading to voltage, for 3.3v arduino use 3.3
  float voltage = reading * 5.0;
  voltage /= 1024.0;

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

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

  // now convert to Fahrenheit
```

```

float temperatureF = (temperatureC * 9.0 / 5.0) + 32.0;
Serial.print(temperatureF); Serial.println(" degrees F");

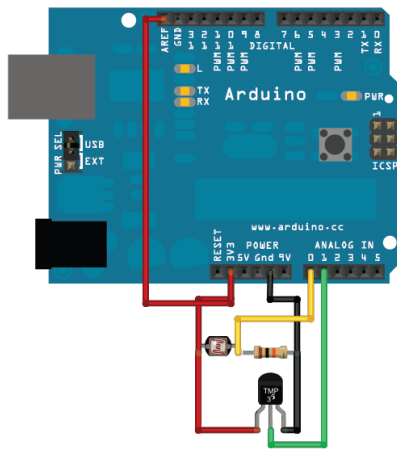
delay(1000);           //waiting a second
}

```

Getting Better Precision

For better results, using the 3.3v reference voltage as ARef instead of the 5V will be more precise and less noisy

This example from the light&temp datalogging tutorial has a photocell but you can ignore it
Note we've changed the TMP36 to A1



```

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

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

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

void setup(void) {

```

```

17 // We'll send debugging information via the Serial monitor
18 Serial.begin(9600);
19
20 // If you want to set the aref to something other than 5v
21 analogReference(EXTERNAL);
22 }
23
24
25 void loop(void) {
26
27   tempReading = analogRead(tempPin);
28
29   Serial.print("Temp reading = ");
30   Serial.print(tempReading); // the raw analog reading
31
32   // converting that reading to voltage, which is based off the reference voltage
33   float voltage = tempReading * aref_voltage;
34   voltage /= 1024.0;
35
36   // print out the voltage
37   Serial.print(" - ");
38   Serial.print(voltage); Serial.println(" volts");
39
40   // now print out the temperature
41   float temperatureC = (voltage - 0.5) * 100 ; //converting from 10 mv per degree wit 500 mV
offset
42                                     //to degrees ((voltage - 500mV) times 100)
43   Serial.print(temperatureC); Serial.println(" degrees C");
44
45   // now convert to Fahrenheit
46   float temperatureF = (temperatureC * 9.0 / 5.0) + 32.0;
47   Serial.print(temperatureF); Serial.println(" degrees F");
48
49   delay(1000);
50 }

```

Ultrasonic sensor interfacing with Aurdino

The HC-SR04 ultrasonic sensor uses SONAR to determine the distance of an object just like the bats do. It offers excellent non-contact range detection with high accuracy and stable readings in an easy-to-use package from 2 cm to 400 cm or 1" to 13 feet.

The operation is not affected by sunlight or black material, although acoustically, soft materials like cloth can be difficult to detect. It comes complete with ultrasonic transmitter and receiver module.

Technical Specifications

- Power Supply – +5V DC
- Quiescent Current – <2mA
- Working Current – 15mA
- Effectual Angle – <15°
- Ranging Distance – 2cm – 400 cm/1" – 13ft
- Resolution – 0.3 cm
- Measuring Angle – 30 degree

Components Required

You will need the following components –

- 1 × Breadboard
- 1 × Arduino Uno R3

- 1 × ULTRASONIC Sensor (HC-SR04)

Procedure

Follow the circuit diagram and make the connections as shown in the image given below.

Sketch

Open the Arduino IDE software on your computer. Coding in the Arduino language will control your circuit. Open a new sketch File by clicking New.

Arduino Code

```
const int pingPin = 7; // Trigger Pin of Ultrasonic Sensor
const int echoPin = 6; // Echo Pin of Ultrasonic Sensor

void setup() {
  Serial.begin(9600); // Starting Serial Terminal
}

void loop() {
  long duration, inches, cm;
  pinMode(pingPin, OUTPUT);
  digitalWrite(pingPin, LOW);
  delayMicroseconds(2);
  digitalWrite(pingPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(pingPin, LOW);
  pinMode(echoPin, INPUT);
```



```
duration = pulseIn(echoPin, HIGH);
inches = microsecondsToInches(duration);
cm = microsecondsToCentimeters(duration);
Serial.print(inches);
Serial.print("in, ");
Serial.print(cm);
Serial.print("cm");
Serial.println();
delay(100);
}

long microsecondsToInches(long microseconds) {
    return microseconds / 74 / 2;
}

long microsecondsToCentimeters(long microseconds) {
    return microseconds / 29 / 2;
}
```

Code to Note

The Ultrasonic sensor has four terminals - +5V, Trigger, Echo, and GND connected as follows –

- Connect the +5V pin to +5v on your Arduino board.
- Connect Trigger to digital pin 7 on your Arduino board.
- Connect Echo to digital pin 6 on your Arduino board.
- Connect GND with GND on Arduino.

In our program, we have displayed the distance measured by the sensor in inches and cm via the serial port.

Result

You will see the distance measured by sensor in inches and cm on Arduino serial monitor.

RF module with Aurdino Controller

Transmitter :

Working voltage: 3V - 12V fo max. power use 12V

Working current: max Less than 40mA max , and min 9mA

Resonance mode: (SAW)

Modulation mode: ASK

Working frequency: Eve 315MHz Or 433MHz

Transmission power: 25mW (315MHz at 12V)

Frequency error: +150kHz (max)

Velocity : less than 10Kbps

So this module will transmit up to 90m in open area .

Receiver :

Working voltage: 5.0VDC +0.5V

Working current: $\leq 5.5\text{mA}$ max

Working method: OOK/ASK

Working frequency: 315MHz-433.92MHz

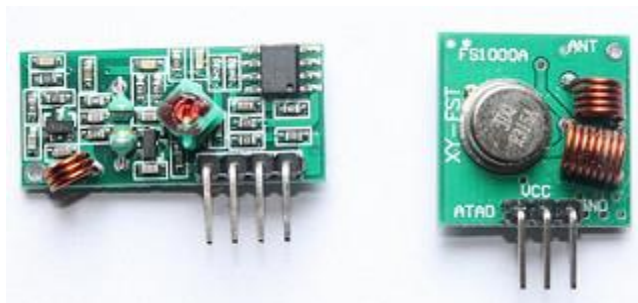
Bandwidth: 2MHz

Sensitivity: excel -100dBm (50Ω)

Transmitting velocity: $< 9.6\text{Kbps}$ (at 315MHz and -95dBm)

Description: This wireless transmitter and receiver pair operate at 315Mhz. They can easily fit into a breadboard and work well with microcontrollers to create a very simple wireless data link. Since these are only transmitters, they will only work communicating data one-way, you would need two pairs (of different frequencies) to act as a transmitter/receiver pair.

Note: These modules are indiscriminate and will receive a fair amount of noise. Both the transmitter and receiver work at common frequencies and don't have IDs. Therefore, a method of filtering this noise and pairing transmitter and receiver will be necessary. The example code below shows such an example for basic operation. Please refer to the example code and links below for ways to accomplish a robust wireless data link.



Module on the right: Transmitter

Module on the left: Receiver

Application

environment

Remote control switch, receiver module, motorcycles, automobile anti-theft products, home security products, electric doors, shutter doors, windows, remote control socket, remote control LED, remote audio remote control electric doors, garage door remote control, remote control retractable doors, remote volume gate, pan doors, remote control door opener, door closing device control system, remote control curtains, alarm host, alarm, remote control motorcycle remote control electric cars, remote control MP3.

Receiver module parameters



1. Product Model: MX-05V
2. Operating voltage: DC5V
3. Quiescent Current: 4mA
4. Receiving frequency: 315Mhz
5. Receiver sensitivity: -105DB
6. Size: 30 * 14 * 7mm

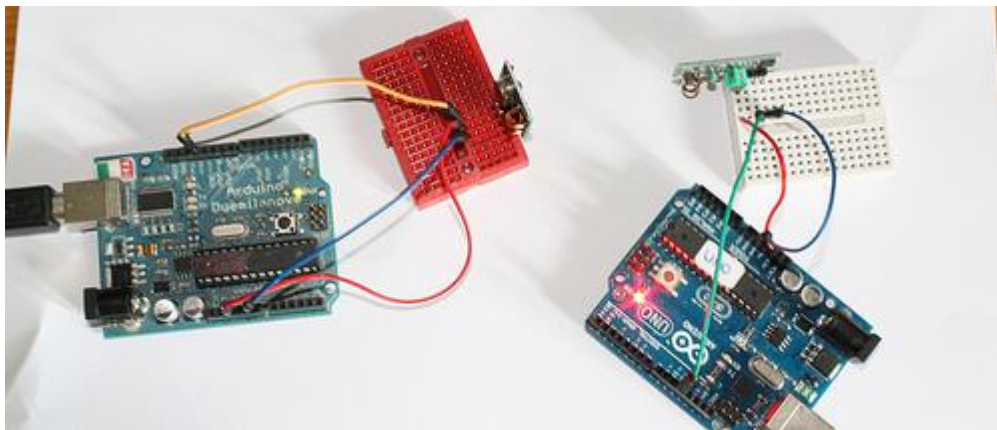
Technical parameters of the transmitter module



1. Product Model: MX-FS-03V

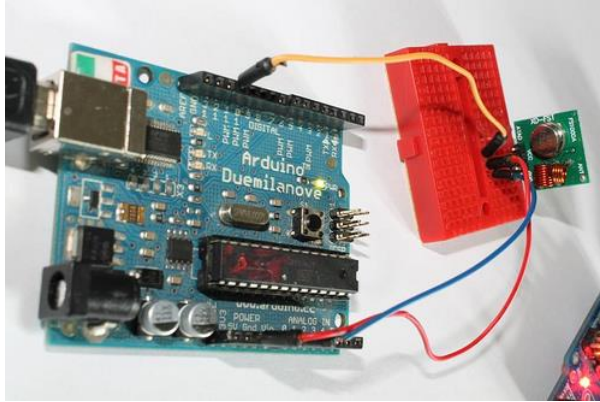
2. Launch distance :20-200 meters (different voltage, different results)
3. Operating voltage :3.5-12V
4. Dimensions: 19 * 19mm
5. Operating mode: AM
6. Transfer rate: 4KB / S
7. Transmitting power: 10mW
8. Transmitting frequency: 315Mhz
9. An external antenna: 25cm ordinary multi-core or single-core line
10. Pinout from left to right: (DATA; VCC; GND)

Here , receiver and transmitter modules are connected separately to two Arduino boards. The transmitter data pin is connected to Pin 12 of Arduino and the receiver data pin is connected to Pin 11 of Arduino.



Data pin of transmitter module to Pin 12 of Arduino.
Data pin of receiver module to Pin 11 of Arduino.
Please note that there are two separate Arduinos for each module.

Connecting transmitter module to Arduino:



Sketch:

```
/*  
SimpleSend  
This sketch transmits a short text message using the VirtualWire library  
connect the Transmitter data pin to Arduino pin 12  
*/  
#include <VirtualWire.h>  
void setup()  
{  
  // Initialize the IO and ISR  
  vw_setup(2000); // Bits per sec  
}  
void loop()  
{  
  send("Hello there");  
  delay(1000);  
}
```

```
void send (char *message)
{
  vw_send((uint8_t *)message, strlen(message));
  vw_wait_tx(); // Wait until the whole message is gone
}
```

Analog Input

In this example we use a variable resistor (a potentiometer or a photoresistor), we read its value using one analog input of an Arduino or Genuino board and we change the blink rate of the built-in LED accordingly. The resistor's analog value is read as a voltage because this is how the analog inputs work.

Hardware Required

- Arduino or Genuino Board
- Potentiometer *or*
- 10K ohm photoresistor and 10K ohm resistor
- built-in LED on pin 13 *or*
- 220 ohm resistor and red LED

Circuit

With a potentiometer click the image to enlarge

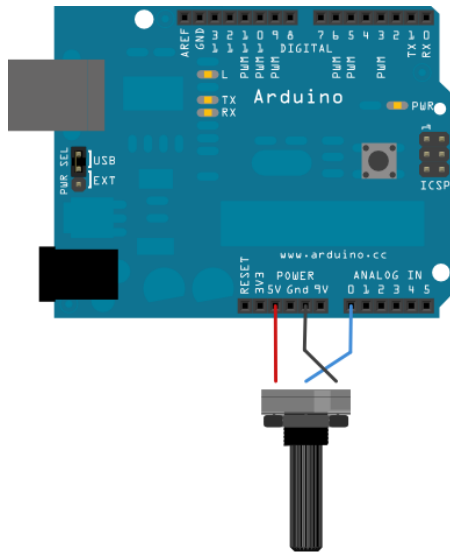


image developed using [Fritzing](#). For more circuit examples, see the [Fritzing project page](#)

With a photoresistor click the image to enlarge

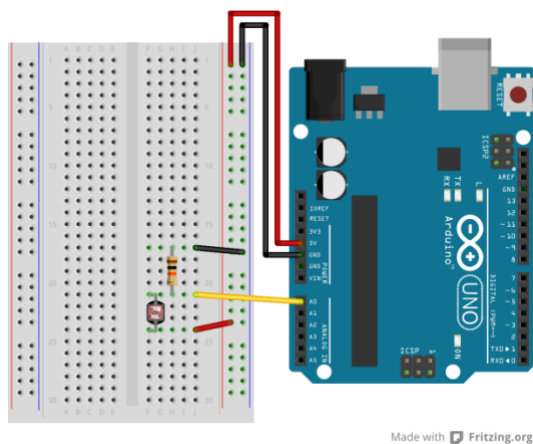


image developed using [Fritzing](#). For more circuit examples, see the [Fritzing project page](#)

Connect three wires to the Arduino or Genuino board. The first goes to ground from one of the outer pins of the potentiometer. The second goes from 5 volts to the other outer pin of the potentiometer. The third goes from analog input 0 to the middle pin of the potentiometer.

For this example, it is possible to use the board's built in LED attached to pin 13. To use an additional LED, attach its longer leg (the positive leg, or anode), to digital pin 13 in series with the 220 ohm resistor, and its shorter leg (the negative leg, or cathode) to the ground (GND) pin next to pin 13.

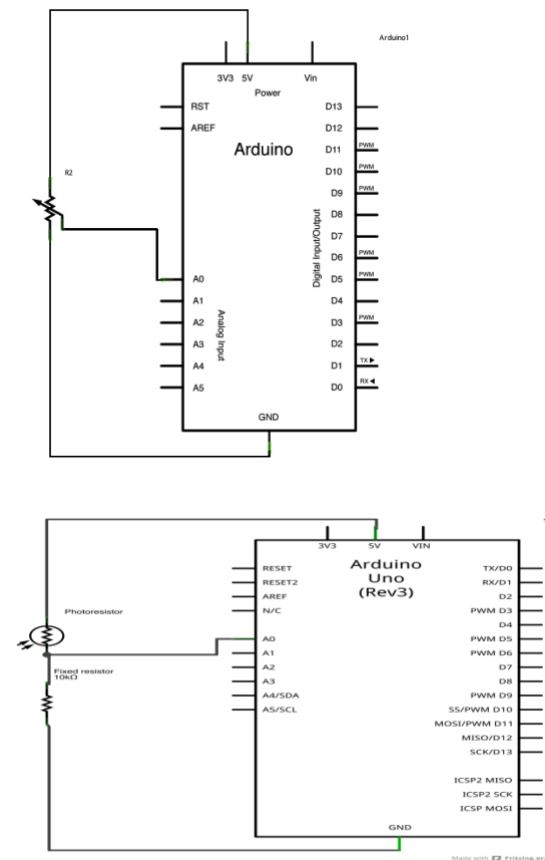
The circuit based on a photoresistor uses a resistor divider to allow the high impedance Analog input to measure the voltage. These inputs do not draw almost any current, therefore by Ohm's law the voltage measured on the other end of a resistor connected to 5V is always 5V, regardless the resistor's value. To get a voltage proportional to the photoresistor value, a resistor divider is necessary. This circuit uses a variable resistor, a fixed resistor and the measurement point is in the middle of the resistors. The voltage measured (V_{out}) follows this formula:

$$V_{out} = V_{in} * (R_2 / (R_1 + R_2))$$

where V_{in} is 5V, R_2 is 10k ohm and R_1 is the photoresistor value that ranges from 1M ohm in darkness to 10k ohm in daylight (10 lumen) and less than 1k ohm in bright light or sunlight (>100 lumen).

Schematic

click [the image](#) to [enlarge](#)



Code

At the beginning of this sketch, the variable `sensorPin` is set to analog pin 0, where your potentiometer is attached, and `ledPin` is set to digital pin 13. You'll also create another variable, `sensorValue` to store the values read from your sensor.

The `analogRead()` command converts the input voltage range, 0 to 5 volts, to a digital value between 0 and 1023. This is done by a circuit inside the microcontroller called an *analog-to-digital converter* or ADC.

By turning the shaft of the potentiometer, you change the amount of resistance on either side of the center pin (or wiper) of the potentiometer. This changes the relative resistances between the center pin and the two outside pins, giving you a different voltage at the analog input. When the shaft is turned all the way in one direction, there is no resistance between the center pin and the pin connected to ground. The voltage at the center pin then is 0 volts, and `analogRead()` returns 0. When the shaft is turned all the way in the other direction, there is no resistance between the center pin and the pin connected to +5 volts. The voltage at the center pin then is 5 volts, and `analogRead()` returns 1023. In between, `analogRead()` returns a number between 0 and 1023 that is proportional to the amount of voltage being applied to the pin.

That value, stored in `sensorValue`, is used to set a `delay()` for your blink cycle. The higher the value, the longer the cycle, the smaller the value, the shorter the cycle. The value is read at the beginning of the cycle, therefore the on/off time is always equal.

```
/*
                                Analog                                Input
Demonstrates analog input by reading an analog sensor on analog
pin                                0                                and
turning on and off a light emitting diode(LED) connected to
digital                                pin                                13.
The amount of time the LED will be on and off depends on
the                                value                                obtained                                by                                analogRead().

The                                circuit:
* Potentiometer attached to analog input 0
* center pin of the potentiometer to the analog pin
* one side pin (either one) to ground
* the other side pin to +5V
* LED anode (long leg) attached to digital output 13
* LED cathode (short leg) attached to ground

* Note: because most Arduinos have a built-in LED attached
to pin 13 on the board, the LED is optional.
```

Created by David Cuartielles
modified 30 Aug 2011
By Tom Igoe

This example code is in the public domain.

<http://www.arduino.cc/en/Tutorial/AnalogInput>

*/

```
int sensorPin = A0;    // select the input pin for the
                        potentiometer
int ledPin = 13;       // select the pin for the LED
int sensorValue = 0;   // variable to store the value coming from
                        the sensor

void setup() {
  // declare the ledPin as an OUTPUT:
  pinMode(ledPin, OUTPUT);
}

void loop() {
  // read the value from the sensor:
  sensorValue = analogRead(sensorPin);

  // turn the ledPin on
  digitalWrite(ledPin, HIGH);
  // stop the program for <sensorValue> milliseconds:
  delay(sensorValue);
  // turn the ledPin off:
  digitalWrite(ledPin, LOW);
  // stop the program for for <sensorValue> milliseconds:
  delay(sensorValue);
}
```

Internet Interfacing

Important: *IBM Internet of Things Foundation (IoT Foundation) is now named IBM Watson IoT Platform. The Bluemix service names have also changed. This article was written using the previous names. The content and images have not been updated. It is provided "as is." Given the rapid evolution of technology, some steps and illustrations may have changed as well.*

Many of you might live in a technology-saturated home as I do. At the center of this web of connected devices sits my home WiFi router, connected to the Internet through a cable modem. So, when all of a sudden you hear cries from throughout the house because laptops, tablets, game boxes, DVRs, and all the rest of the Internet-connected devices have failed because the WiFi router has stopped working, you know something is amiss.

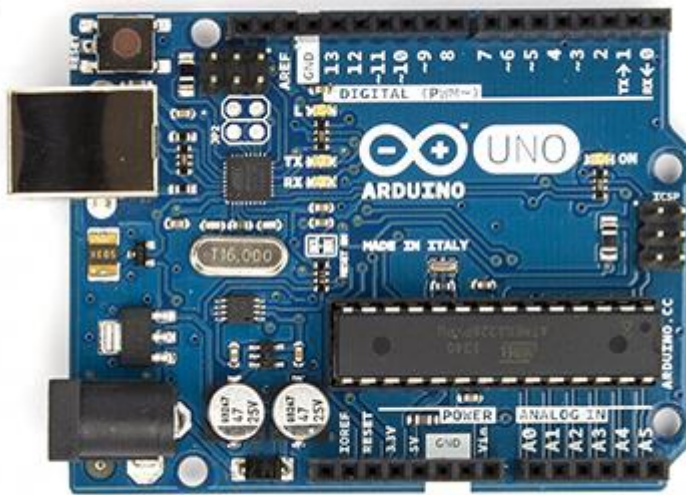
In the past, I've tried multiple approaches to improve this situation: I've replaced the router with a new model, I've updated the firmware, I've replaced the cable modem, and I've had the cable company check the signal on the coax lines — all to no avail. The consensus in the Internet community among those with similar problems is that either the router or the modem (or both) might be getting too hot. That seemed unlikely to me because it doesn't feel warm in the wiring closet where I installed these machines, but my evaluation of the temperatures has not been scientific sampling. What I needed was a way to find out the temperature of the wiring closet when or just before the failures occurred so I could see if there was a correlation. And that need led me to my experiment.

Like many other engineers, I've been entranced for a number of years by the capabilities of the Arduino open source electronics platform. The Arduino is a single-board microcontroller that was introduced in 2005 (and named by its creators after a bar in the Northern Italian town of Ivrea). The Arduino is programmed by using a language (called Processing) that's based on the Java™ language. The Arduino community is enormous, and hundreds of examples of using the board to take measurements from various sensors and control all sorts of actuators exist on the web. So when I thought about my problem, my attention turned to an unused Arduino Uno board sitting beside my desk, together with the Arduino Ethernet shield sitting in my drawer and the brand-new temperature sensor that had just arrived the week before that was waiting for a project. And as I thought about the IBM Internet of Things (IoT) Foundation and the capabilities for IoT development that are part of IBM® Bluemix™, the concept for my project began to take shape. Figure 1 shows the entire project that I wanted to build.

In the design, the Arduino board, connected to a temperature sensor, sends temperature (Fahrenheit and Celsius) and humidity information on a regular schedule to the IoT Foundation via the MQTT protocol and is graphed, using capabilities in Bluemix. The IoT Foundation enables this data to be plotted on a realtime graph. You'll complete this entire project over the course of this series.

“As I thought about the IBM Internet of Things Foundation and the capabilities for IoT development that are part of IBM Bluemix, the concept for my project began to take shape.”

Figure 2. Arduino Uno

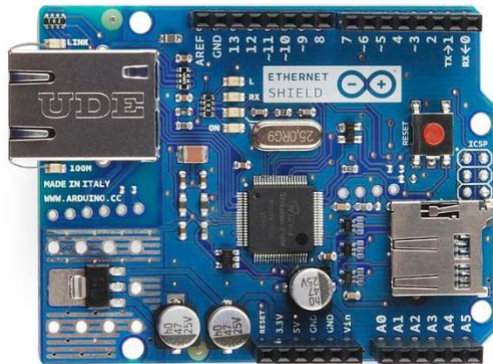


When you first receive your board, visit the [Arduino website](http://www.arduino.cc), which is the home of the Arduino community and contains a wealth of information and documentation on the boards, their capabilities, and the software that has been written to take advantage of them. For this quick "cook's tour," I'll simply start at the top of the board and walk you around the outer edge to review the most important parts of the board from the perspective of this tutorial.

At the very top sits a row of holes for pins that can be used to read values from, or to control digital devices. Moving to the bottom is a row that is half-occupied by pins for reading or writing analog values, while the remaining pins on that row provide different types of power and ground connections. On the bottom left of the board is a jack for a 5V power supply; you'll use this jack if you want to power your project when it's not connected to your computer. The next jack above that on the left side of the board is a connector for a USB cable. This connector is used both for communication to the computer you program the board from, and for powering the board when the cable is connected. Finally, at the top left of the board is the reset button — just the thing for when your software is misbehaving.

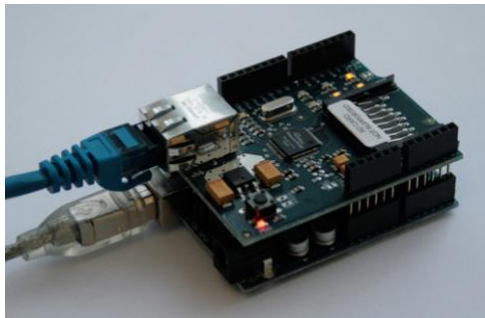
One of the strengths of the Arduino community is the fact that not only has the basic board had its specifications released into the open source community, resulting in many versions of it, but also that many different *shields* have been built for it. The one for this project is the R3 version of the Ethernet shield, which provides access to an Ethernet connection at either 10 or 100MB through a standard RJ45 connector. It's also available from many of the same retailers as the Arduino Uno, for about US\$30. Figure 3 shows the Ethernet shield.

Figure 3. Arduino Ethernet shield



The neat thing about these boards is that one plugs into another — so a project might have an Arduino on the bottom, an Ethernet or WiFi shield in the middle, and perhaps a motor shield (for Robot control) or an additional extension shield for special sensors on the top. You can get a sense of what the boards look like when stacked from the photo in Figure 4.

Figure 4. Arduino Uno and Ethernet shield connected



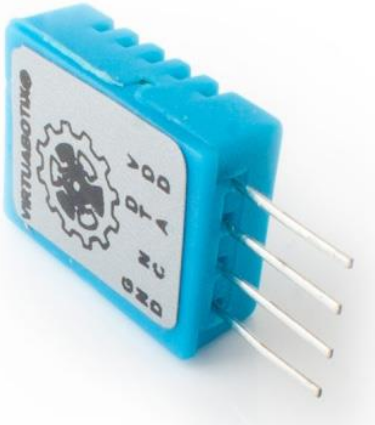
Once you've stacked up the boards, you might feel like you're ready to go, but wait: You must first download and install the Arduino IDE, which is available from the download link at the [Arduino website](http://www.arduino.cc) and is compatible with Mac, Windows®, and Linux®. If you want to, you can try one of the newer beta versions, but for this tutorial I'll use the current released version of the IDE, which is Arduino 1.0.5.

Download the IDE (which usually comes as a ZIP or TAR file, depending on your computer's OS) and follow the instructions to install the IDE and the USB drivers on your computer. At that point, you can then plug in the USB cable to power up the board (and often complete the driver installation), plug in your Ethernet cable, open the IDE, and start trying out the library examples that come with the IDE — but that's getting a bit ahead of the example.

Building the circuit

For this project's purposes, you have a few more pieces of hardware to acquire. The first is a standard prototyping "breadboard." If you're an electrical engineer or electronics hobbyist, you're undoubtedly familiar with these already. If you're not, then you'll need to purchase one like the one shown in Figure 5.

Figure 6. DHT11 temperature and humidity sensor

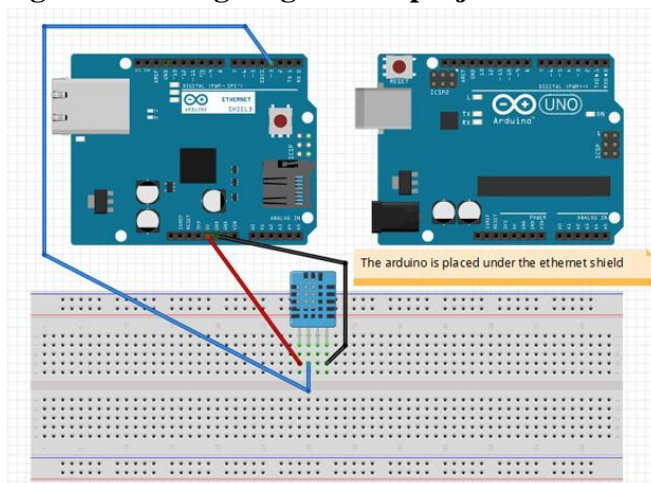


When you receive the DHT11 sensor, it should come with a set of pinout documentation showing what all the pins on the sensor do. If not, you can find that information at the [Virtuabotix website](http://Virtuabotix.com).

While you are visiting the Virtuabotix website, download the Arduino libraries (which come packaged as a ZIP file) for the Virtuabotix DHT11. They are available from the download link on that same page. Extract them into the Libraries subdirectory of the Arduino IDE directory structure. When you restart the Arduino IDE, you'll see that the DHT11 samples are available for use, just like the built-in samples for the Ethernet shield and other devices.

Now that you have all the physical pieces, it's time to wire together your project. The wiring diagram in Figure 7 shows all the connections that you need to make.

Figure 7. Wiring diagram for project

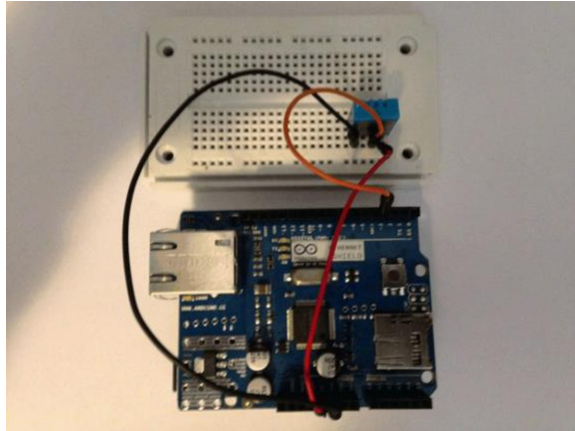


Let's review the wiring of these parts. The Arduino is placed on the bottom, while the Ethernet shield is placed on top of it (see [Figure 4](#) for an example of how this works). I simply plugged the DHT11 into an unused column of pins on the breadboard. Note that the front of the DHT11 with the waffle grille — not the side with the label — is facing forward. The ground pin on the DHT11 (which is the one on the far right side with the waffle grille facing forward) is connected via the black wire to the GND pin on the Arduino. The VDD pin on the DHT11 (the one on the farthest left) is connected via the red wire to the 5V power-supply pin on the Arduino.

Finally, the DTA pin (second from the left) is connected to Digital pin 3 on the Arduino. Remember that pin number —*pin 3*— because that becomes important when you write the software that references the DHT11.

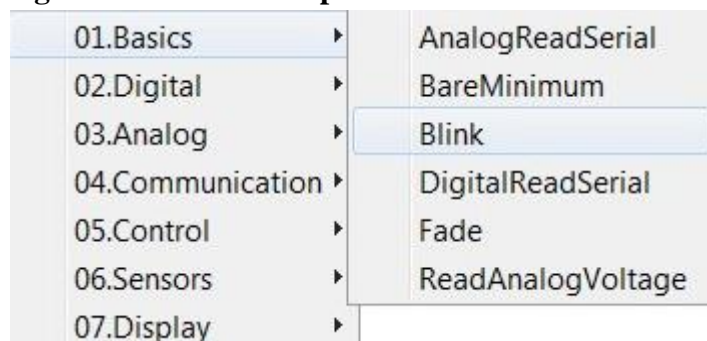
Your final configuration should look something like Figure 8.

Figure 8. Completed wiring for prototype



At this point, it's a good idea to individually test out the components. You might want to start by testing for the proper installation of the USB drivers in your computer to see if it is successfully communicating with the Arduino. Open the Arduino IDE and click **File > Examples > 01.Basics > Blink**, as shown in Figure 9.

Figure 9. Blink menu option



The `Blink` sketch (programs in the Arduino are called *sketches*) is a simple one that enables you to test communication to the Arduino. All it does is cause one of the built-in LEDs on the Arduino board to blink on and off with a period of one second. If you correctly installed the USB device drivers, you should be able to select your board type by clicking **Tools > Board** (for instance, if you're using a Uno, click that) and your serial Port from the SerialPort menu (refer to your OS tools such as the Windows Device Manager for the name or number of the serial port you are using). If either of these selections is unavailable, refer to the help on the Arduino website. After you select your board and your serial port, you can upload the sketch to the board by clicking the **Upload** button (the right-pointing arrow, second button from the left in the toolbar), as shown in Figure 10.

Figure 10. Blink sketch with upload button



If the message `Done uploading` is displayed and you see the LED begin to blink, then you're successfully in communication with the Arduino. If not, again, refer to the [Arduino website](#) for help. Once you've proven that you can connect to the Arduino, you should try one or more of the Ethernet shield examples, by clicking **Examples > Ethernet > Web Server**. A trick for making the web server example functional is that it uses two parameters that you probably need to change. Near the top of the example sketch, you'll see the following lines of code:

```
1 // Enter a MAC address and IP address for your controller below.  
2 // The IP address will be dependent on your local network:  
3 byte mac[] = {  
4   0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };  
5 IPAddress ip(192,168,1,177);
```

On the newer Ethernet shields, the MAC address is on a sticker attached to the shield. (For older boards, you'll need to make one up.) Change the `mac[]` variable values in the code to match the MAC address shown on the sticker. Likewise, you need to change the IP address to an open address on your local network. Make these changes, save the sketch to a new name, and then upload it. It should work now. Finally, if you downloaded and extracted the DHT11 libraries to the correct directory, you should be able to click **Examples > DHT11 > dht11_functions** and load that sketch. Near the top of that sketch, you'll see the following code:

```
1 void setup()
2 {
3   DHT11.attach(2);
4   Serial.begin(9600);
5   Serial.println("DHT11 TEST PROGRAM ");
6   Serial.print("LIBRARY VERSION: ");
7   Serial.println(DHT11LIB_VERSION);
8 }
```

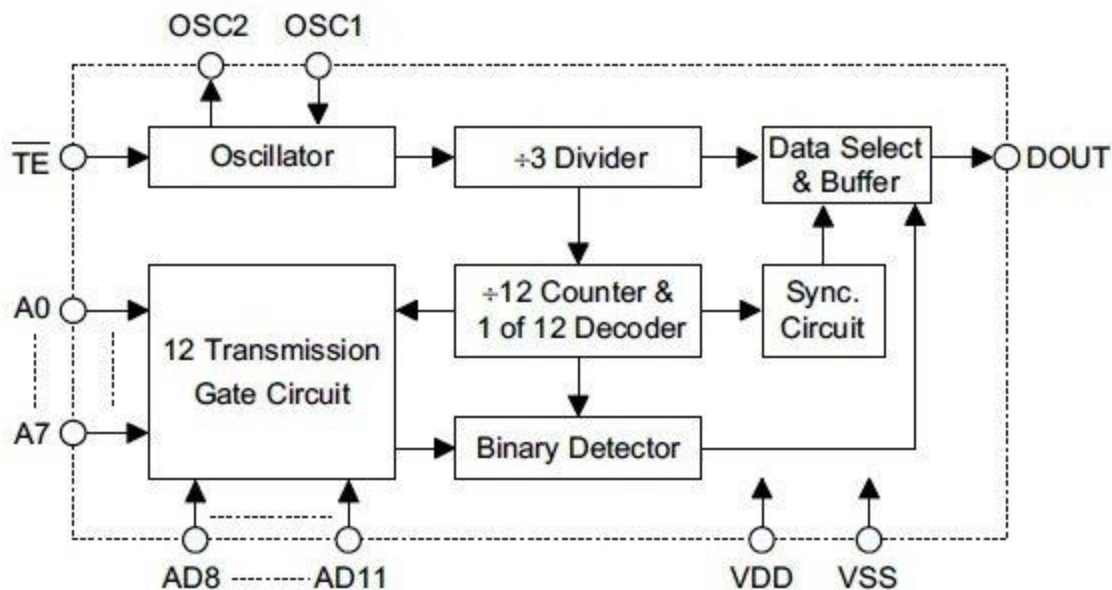
Remember that you attached the DHT11 DTA pin to pin 3 on the Arduino? That's because pin 2 is used by the Ethernet shield and is unavailable. So change the 2 in the line `DHT11.attach(2)` to a 3, save the sketch to a different name, and then upload it. Then open the Serial Monitor (click **Tools > Serial Monitor**) and you should see output from the program showing temperature and humidity data scrolling down the screen.

Conclusion and next steps

Now that you've proven that all of the parts are functional, you're ready to move on to the next part of the series. In [Part 2](#), I explain the communications protocol of the IoT — MQTT — and discuss the Arduino client libraries for MQTT as well as some hints and techniques for testing your MQTT programs locally. I also discuss the capabilities of the IBM IoT Foundation, show you how to connect a program running on the Arduino into the IoT Foundation, and bring all the pieces together for showing how the sensor data comes in, is formatted for the IoT, and is sent to the cloud.

Serial & parallel Communication

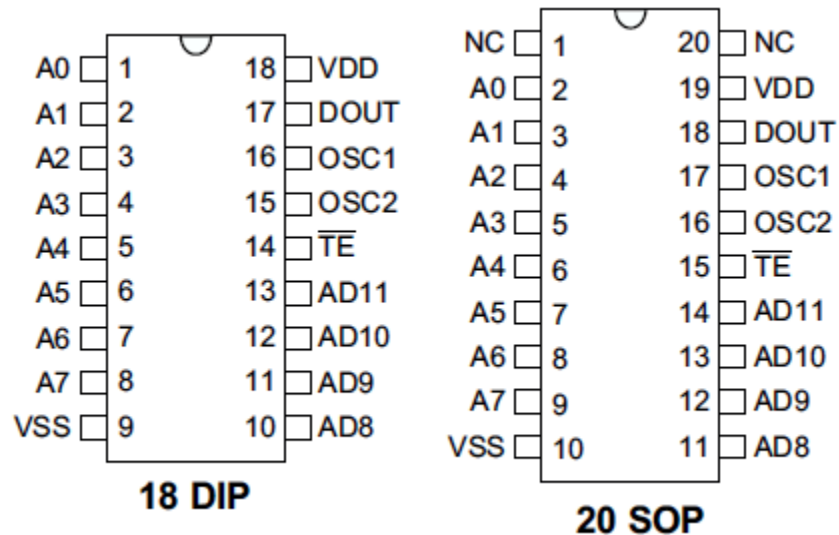
HT12E is a 2^{12} series encoder IC (Integrated Circuit) for remote control applications. It is commonly used for radio frequency (RF) applications. By using the paired HT12E encoder and [HT12D](#) decoder we can easily transmit and receive 12 bits of parallel data serially. HT12E simply converts 12 bit parallel data in to serial output which can be transmitted through a RF transmitter. These 12 bit parallel data is divided in to 8 address bits and 4 data bits. By using these address pins we can provide 8 bit security code for data transmission and multiple receivers may be addressed using the same transmitter.



HT12E – Block Diagram

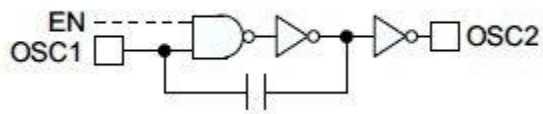
HT12E is able to operate in a wide voltage range from 2.4V to 12V and has a built in oscillator which requires only a small external resistor. Its power consumption is very low, standby current is $0.1\mu\text{A}$ at 5V VDD and has high immunity against noise. It is available in 18 pin DIP (Dual Inline Package) and 20 pin SOP (Small Outline Package) as given below.

Pin Diagram and Description



HT12E – Pin Diagram

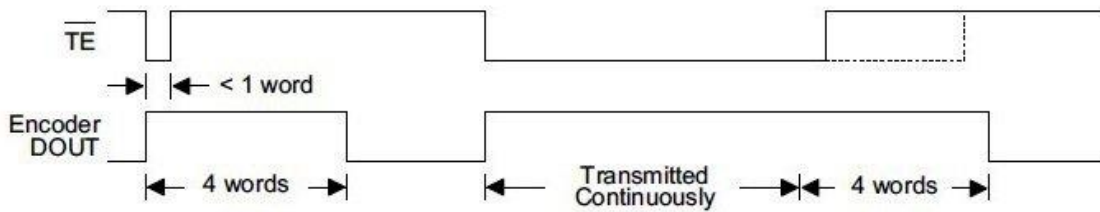
- **VDD and VSS** are power supply pins which are used to connect positive and negative of the power supply respectively.
- **OSC1 and OSC2** are used to connect external resistance for the internal oscillator. OSC1 is the oscillator input pin and OSC2 is the oscillator output pin.



Oscillator of HT12E

- **TE** is used for enabling the transmission and is an active low input.
- **A0 – A7** are the input address pins. By using these pins we can provide a security code for the data. These pins can be connected to VSS or left open.
- **D8 – D11** are the input data pins. These pins can be connected to VSS or may left open for sending LOW and HIGH respectively.
- **DOUT** – It is the serial data output of the encoder and can be connected to a RF transmitter.

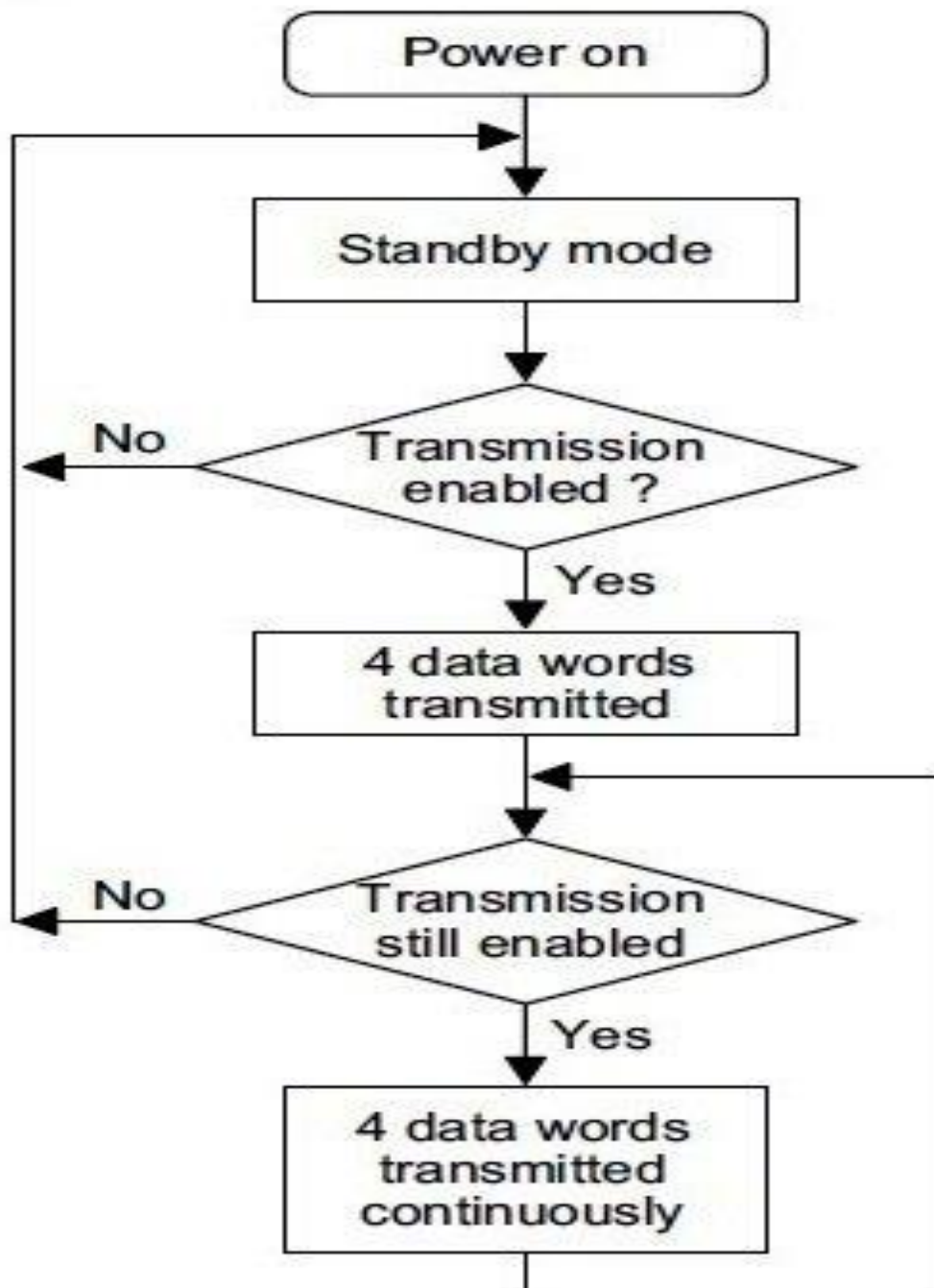
Working



Transmission timing for the HT12E

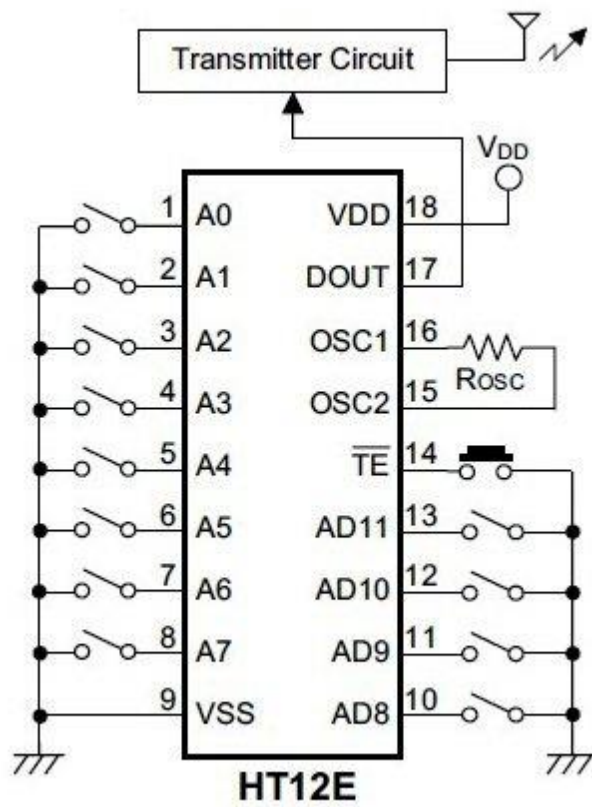
Transmission Timing for HT12E

The HT12E 2¹² series encoder starts a 4 word transmission cycle upon receiving transmission enable signal on TE input. This output cycle will repeats as long as the transmission is enabled. When the transmission enable (TE) signal switches to HIGH, the encoder output completes the current cycle and stops as shown above. The encoder will be in the Standby mode when the transmission is disabled.



Working Flowchart of HT12E

Typical Application Circuit



Ethernet Shield

The Arduino Ethernet Shield 2 connects your Arduino to the internet in mere minutes. Just plug this module onto your Arduino Board, connect it to your network with an RJ45 cable (not included) and follow a few simple steps to start controlling your world canadian pharmacy through the internet. As always with Arduino, every element of the platform – hardware, software and documentation – is freely available and open-source. This means you can learn exactly how it's made and use its design as the starting point for your own circuits. Hundreds of thousands of Arduino Boards are already fueling people's creativity all over the world, everyday. Join us now, Arduino is you!

*Requires an Arduino Board (not included)

- Operating voltage 5V (supplied from the Arduino Board)
- Ethernet Controller: W5500 with internal 32K buffer
- Connection speed: 10/100Mb

- Connection with Arduino on SPI port
- On the Ethernet Shield 2 [on the Ethernet 2 Library](#)
- On Projects [on the Arduino Forum](#)
- On the Product itself through [our Customer Support](#)

Technical Specs

The Arduino Ethernet Shield 2 allows an Arduino Board to connect to the internet. It is based on the ([Wiznet W5500 Ethernet chip](#)). The Wiznet W5500 provides a network (IP) stack capable of both TCP and UDP. It supports up to eight simultaneous socket connections. Use the Ethernet library to write sketches that connect to the Internet using the Shield. The Ethernet Shield 2 connects to an Arduino Board using long wire-wrap headers extending through the Shield. This keeps the pin layout intact and allows another Shield to be stacked on top of it.

The most recent revision of the board exposes the 1.0 pinout on rev 3 of the Arduino UNO Board.

The Ethernet Shield 2 has a standard RJ-45 connection, with an integrated line transformer and Power over Ethernet enabled.

There is an onboard micro-SD card slot, which can be used to store files for serving over the network. It is compatible with the Arduino Uno and Mega (using the Ethernet library). The onboard micro-SD card reader is accessible through the SD Library. When working with this library, SS is on Pin 4. The original revision of the Shield contained a full-size SD card slot; this is not supported.

The Shield also includes a reset controller, to ensure that the W5500 Ethernet module is properly reset on power-up. Previous revisions of the Shield were not compatible with the Mega and needed to be manually reset after power-up. The current Shield has a Power over Ethernet (PoE) module designed to extract power from a conventional twisted pair Category 5 Ethernet cable. PoE module features as follows:

- IEEE802.3af compliant
- Input voltage range 36V to 57V
- Overload and short-circuit protection
- 12V Output
- High efficiency DC/DC converter: typ 85% @ 80% load
- 1500V isolation (input to output)

NB: the Power over Ethernet module is proprietary hardware not made by Arduino, it is a third party accessory. For more information, see the [datasheet](#)

The Shield does not come with a built in PoE module, it is a separate component that must be added on. Arduino communicates with both the W5500 and SD card using the SPI bus (through the ICSP header). This is on digital pins 10, 11, 12, and 13 on the Uno and pins 50, 51, and 52 on the Mega. On both boards, pin 10 is used to select the W5500 and pin 4 for the SD card. These pins cannot be used for general I/O. On the Mega, the hardware SS pin, 53, is not used to select either the W5500 or the SD card, but it must be kept as an output or the SPI interface won't work.

Note that because the W5500 and SD card share the SPI bus, only one at a time can be active. If you are using both peripherals in your program, this should be taken care of by the corresponding libraries. If you're not using one of the peripherals in your program, however, you'll need to explicitly deselect it. To do this with the SD card, set pin 4 as an output and write a high to it. For the W5500, set digital pin 10 as a high output.

- The Shield provides a standard RJ45 Ethernet jack.
- The reset button on the Shield resets both the W5500 and the Arduino Board.
- The Shield contains a number of information LEDs:
 - ON: indicates that the Board and Shield are powered
 - LINK: indicates the presence of a network link and flashes when the Shield transmits or receives data
 - FDX: indicates that the network connection is full duplex
 - 100M: indicates the presence of a 100 Mb/s network connection (as opposed to 10 Mb/s)
 - ACT: flashes when RX or TX activity is present

Initializing Micro-SD card on an Ethernet shield



The latest Arduino Ethernet shield comes with a handy MicroSD card slot so that you can store and retrieve data through the shield. Very handy! Lets show how to talk to the card.

Be sure to have the very latest version of SdFatLib , as you'll need some of the newer capabilities!

First thing to note is that the **SS** (Slave Select) pin for the card is **digital 4** (although as of the writing of this mini-tutorial, theschematic hasn't been updated, you'll have to trust me!)

Open up the **SdFatInfo** example sketch and change the line in **loop()** from

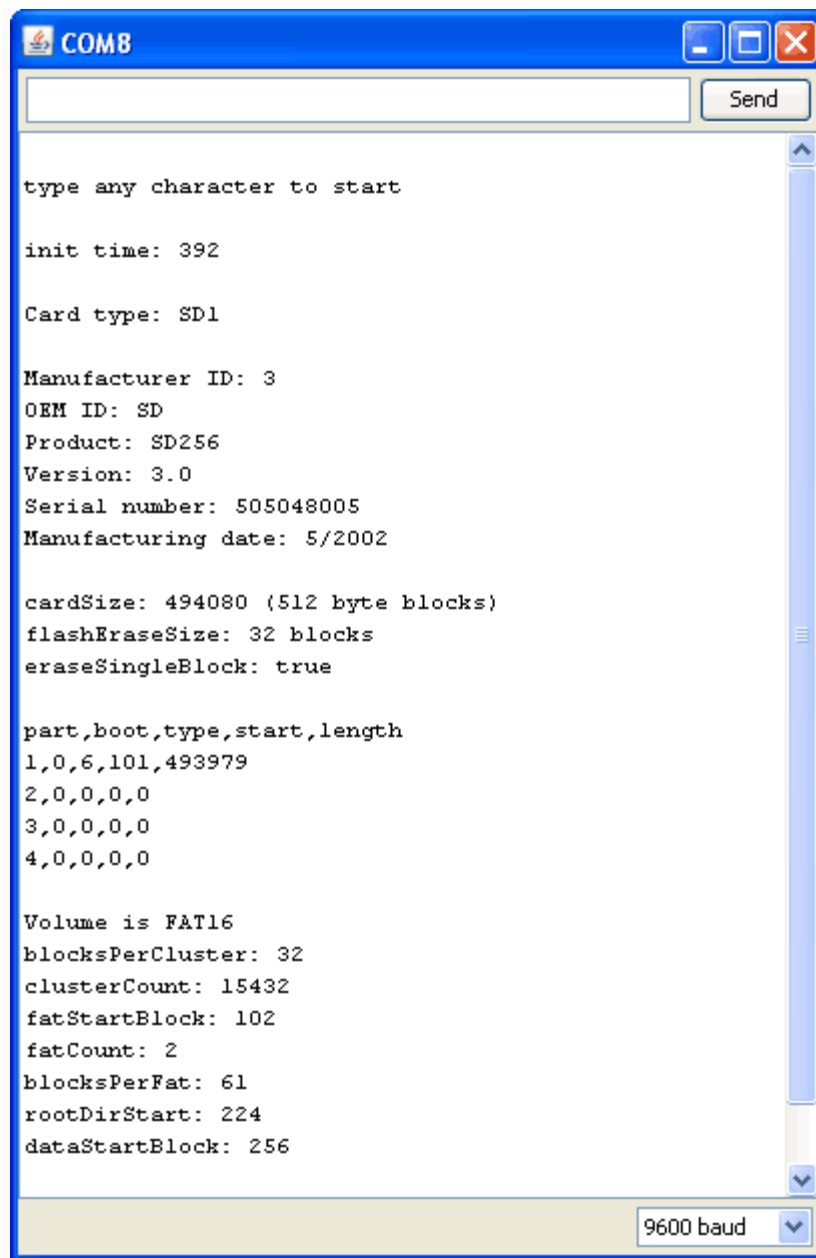
```
uint8_t r = card.init(SPI_HALF_SPEED);
```

To:

```
pinMode(10, OUTPUT);           // set the SS pin as an output (necessary!)
digitalWrite(10, HIGH);        // but turn off the W5100 chip!
uint8_t r = card.init(SPI_HALF_SPEED, 4); // Use digital 4 as the SD SS line
```

Be sure to add those two extra lines right before-hand! They Enable the SPI interface. If you're on a Mega, use pin **53** instead of **10**

Now upload and test the card, you should see something like this:

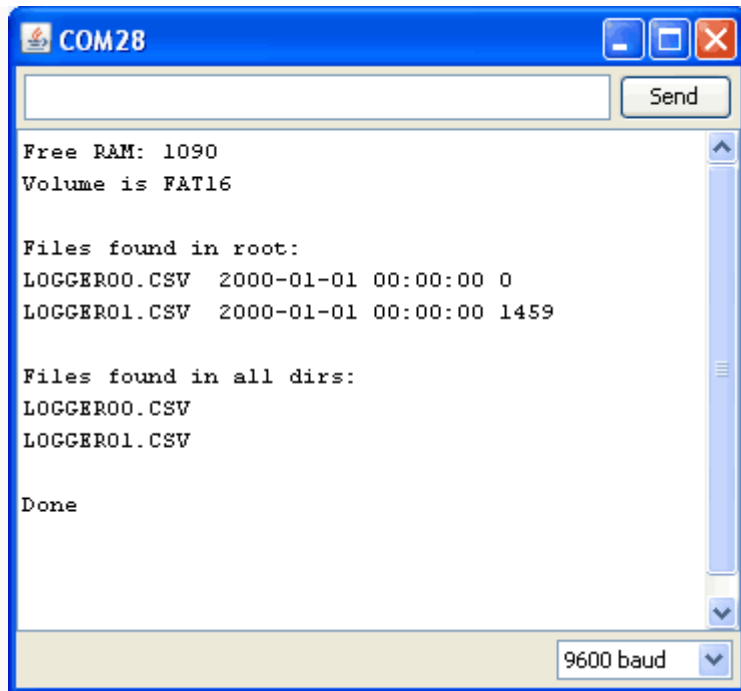


Indicating you talked to the card all right

List files

Put some text files on your SD card, using a computer, so that you have data to read. Make sure they are in the root directory, and not in a folder

Then run the **SdFatLs** example sketch from the SdFat library, you should see it list all the files you have on the card, again, you'll have to make the changes from above to update the **card.init()** part to the new **SS** pin



For example, the card I'll be using has two files on it from some previous datalogging.

Merge WebServer + SdFatLs

We'll begin by combining WebServer (the example sketch that comes with the **Ethernet** lib) and **SdFatLs** to make a web server that lists the files on the SD card. [You can download the file here](#) (you'll need to copy&paste it, do so carefully!) then follow along!

Part one is the Ethernet and SD card objects and a simple error function (**void error_P(const char* str)**) that prints out errors and halts the program if there are serious problems.

You should, of course, set your **mac** and **ip** as necessary, use the **ip** and **mac** that worked from your previous Ethernet shield explorations!

The **card**, **volume**, and **root** are objects that help us traverse the complex structure of an SD card

The error function is not too exciting, it just prints out the error and sits in a **while(1);** loop forever

```
/*  
  
 * This sketch will list all files in the root directory and  
 * then do a recursive list of all directories on the SD card.  
 *  
 */
```

```
#include <SdFat.h>

#include <SdFatUtil.h>

#include <Ethernet.h>

/%%***%%**%%%**%%**%%%**%%**%%%**%%**%%**%% ETHERNET STUFF
%%***%%**%%%**%%**%%%**%%**%%%**%%**%%**%%/

byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };

byte ip[] = { 192, 168, 1, 177 };

Server server(80);

/%%***%%**%%%**%%**%%%**%%**%%%**%%**%%**%% SDCARD STUFF
%%***%%**%%%**%%**%%%**%%**%%%**%%**%%**%%/

Sd2Card card;

SdVolume volume;

SdFile root;

// store error strings in flash to save RAM

#define error(s) error_P(PSTR(s))

void error_P(const char* str) {

  PgmPrint("error: ");

  SerialPrintln_P(str);

  if (card.errorCode()) {

    PgmPrint("SD error: ");

    Serial.print(card.errorCode(), HEX);

    Serial.print(',');

    Serial.println(card.errorData(), HEX);
```

```

}

while(1);

}

```

Part 2 is the **setup()** function. It sets up the Serial object so we can debug the connection in real time. It then prints out the RAM usage. You'll need a **Atmega328** Arduino for this experiment, and you should see at least 1000 bytes of RAM free. Once this gets to under 250 bytes, you may be running too low!

Then we do the trick where we make the hardware **SS** pin #10 to an **OUTPUT** and **HIGH** to disable the wiznet chip while we check the card contents. If you're on a Mega, change this to 53. Then we initialize the card which should go fine since you already tested this before

Then we verify the card structure, print out all the files, and print "Done!". Finally we stuck the Ethernet initialization code at the end here! Now we have both the Ethernet and SD card working

```

void setup() {

  Serial.begin(9600);

  PgmPrint("Free RAM: ");
  Serial.println(FreeRam());

  // initialize the SD card at SPI_HALF_SPEED to avoid bus errors with
  // breadboards.  use SPI_FULL_SPEED for better performance.

  pinMode(10, OUTPUT);                // set the SS pin as an output
  (necessary!)

  digitalWrite(10, HIGH);              // but turn off the W5100 chip!

  if (!card.init(SPI_HALF_SPEED, 4)) error("card.init failed!");

  // initialize a FAT volume

  if (!volume.init(&card)) error("vol.init failed!");

  PgmPrint("Volume is FAT");
}

```

```

Serial.println(volume.fatType(),DEC);

Serial.println();

if (!root.openRoot(&volume)) error("openRoot failed");

// list file in root with date and size

PgmPrintln("Files found in root:");

root.ls(LS_DATE | LS_SIZE);

Serial.println();

// Recursive list of all directories

PgmPrintln("Files found in all dirs:");

root.ls(LS_R);

Serial.println();

PgmPrintln("Done");

// Debugging complete, we start the server!

Ethernet.begin(mac, ip);

server.begin();

}

```

We'll skip ahead to the **loop()** where we wait for clients (checking via **server.available()**) and then read the client request before responding. This is basically copy-and-pasted from the **Webserver** example sketch that comes with the Ethernet library (well, the first and last parts of the loop are at least).

There's a little trick where to simplify the code, the writer of this sketch doesn't actually check to see what file the web browser wants, it always spits out the same thing. In this case, we're going to have it spit out the files by using a helper function called **ListFiles(client, 0)** which we skipped over but will show next. The 0 in the second argument to the function just tells the function whether to print out the file sizes


```
void loop()
{
  Client client = server.available();

  if (client) {

    // an http request ends with a blank line

    boolean current_line_is_blank = true;

    while (client.connected()) {

      if (client.available()) {

        char c = client.read();

        // if we've gotten to the end of the line (received a newline
        // character) and the line is blank, the http request has ended,
        // so we can send a reply

        if (c == '\n' && current_line_is_blank) {

          // send a standard http response header

          client.println("HTTP/1.1 200 OK");

          client.println("Content-Type: text/html");

          client.println();

          // print all the files, use a helper to keep it clean
          //ListFiles(client, 0);

          client.println("<h2>Files:</h2>");

          ListFiles(client, 0);

          break;

        }

        if (c == '\n'){

          // we're starting a new line
```

```

        current_line_is_blank = true;

    } else if (c != '\r') {

        // we've gotten a character on the current line

        current_line_is_blank = false;

    }

}

}

}

// give the web browser time to receive the data

delay(1);

client.stop();

}

}

```

Now we'll go back and examine the **ListFiles** function. This is a bit tedious, but worth looking at. We've simplified it by removing recursive listing, which means we don't list files in any subdirectories.

The **dir_t** object is a "Directory Entry" holder. It will store the information for each entry in the directory.

We first reset the root directory by **rewind()**'ing it. Then we read the directory file by file. Some files are unused or are the "." and ".." (up directory) links, which we ignore. We also only list FILES or SUBDIRectories.

Then we print the name out by going through all 11 characters (remember the file names are in 8.3 format) and ignore the space. We also stick the '.' between the first 8 and last 3 characters.

If its a directory type file, we put a slash at the end to indicate it. If its not, we can print out the file size in bytes.

Finally, after each file name we stick in a "
" which will go to the next line in a web browser

```

void ListFiles(Client client, uint8_t flags) {

    // This code is just copied from SdFile.cpp in the SDFat Library

    // and tweaked to print to the client output in html!

    dir_t p;

```

```

root.rewind ();

while (root.readDir(p) > 0) {

    // done if past last used entry

    if (p.name[0] == DIR_NAME_FREE) break;

    // skip deleted entry and entries for . and ..

    if (p.name[0] == DIR_NAME_DELETED || p.name[0] == '.') continue;

    // only list subdirectories and files

    if (!DIR_IS_FILE_OR_SUBDIR(&p)) continue;

    // print file name with possible blank fill

    //root.printDirName(*p, flags & (LS_DATE | LS_SIZE) ? 14 : 0);

    for (uint8_t i = 0; i < 11; i++) {

        if (p.name[i] == ' ') continue;

        if (i == 8) {

            client.print('.');

        }

        client.print(p.name[i]);

    }

    if (DIR_IS_SUBDIR(&p)) {

        client.print('/');

    }

```

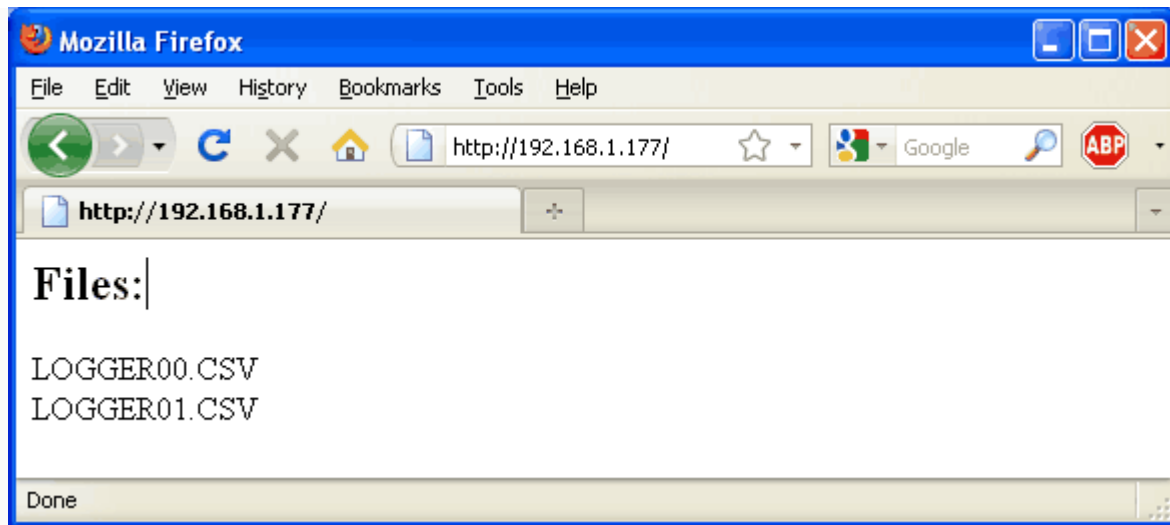
```

// print modify date/time if requested
if (flags & LS_DATE) {
    root.printFatDate(p.lastWriteDate);
    client.print(' ');
    root.printFatTime(p.lastWriteTime);
}

// print size if requested
if (!DIR_IS_SUBDIR(&p) && (flags & LS_SIZE)) {
    client.print(' ');
    client.print(p.fileSize);
}

client.println("<br>");
}
}

```



Browsing files!

Obviously, we should make it so you can clicky those file names, eh? Well! Thats the next sketch, you can [download the latest version from github here](#) (click Download Source) in the top right hand corner!

Fix the **ip** address to match your network, and SS pin (if you're on a Mega)

Not a lot has changed between the previous code, the setup is the same, the big changes are in the **loop()** code. The bones are the same - we look for new client connections. But this time we read the client request into a character buffer (**clientline**) until we get a newline character (such as \n or \r). This indicates we have read an entire line of text. To 'finish' the string, we put a null character (0) at the end.

We then use **strstr** which will look for substrings. If we have a "GET / HTTP" request for the root directory, we do the same as before, printing out the list of files.

If we have no space after "GET /" that means its something like "GET /file" which means we will have to extract the filename. We make a pointer to the string and start it right after the first slash. Then we look for the beginning of the "HTTP/1.1" string which follows the filename request and turn the first character into a string-terminator. Now we have the name of the file which we can try to open.

If we fail to open the file, we will return a **404**. Otherwise, we print out all of the file contents.

```
// How big our line buffer should be. 100 is plenty!

#define BUFSIZ 100

void loop()
{
    char clientline[BUFSIZ];

    int index = 0;

    Client client = server.available();

    if (client) {

        // an http request ends with a blank line

        boolean current_line_is_blank = true;

        // reset the input buffer

        index = 0;

        while (client.connected()) {
```

```
if (client.available()) {  
    char c = client.read();  
  
    // If it isn't a new line, add the character to the buffer  
    if (c != '\n' && c != '\r') {  
        clientline[index] = c;  
        index++;  
        // are we too big for the buffer? start tossing out data  
        if (index >= BUFSIZ)  
            index = BUFSIZ - 1;  
  
        // continue to read more data!  
        continue;  
    }  
  
    // got a \n or \r new line, which means the string is done  
    clientline[index] = 0;  
  
    // Print it out for debugging  
    Serial.println(clientline);  
  
    // Look for substring such as a request to get the root file  
    if (strstr (clientline, "GET / ") != 0) {  
        // send a standard http response header  
        client.println("HTTP/1.1 200 OK");  
        client.println("Content-Type: text/html");  
        client.println();  
    }  
}
```

```

    // print all the files, use a helper to keep it clean

    client.println("<h2>Files:</h2>");

    ListFiles(client, LS_SIZE);

} else if (strstr (clientline, "GET /") != 0) {

    // this time no space after the /, so a sub-file!

    char *filename;

    filename = clientline + 5; // Look after the "GET /" (5 chars)

    // a little trick, look for the " HTTP/1.1" string and

    // turn the first character of the substring into a 0 to clear it out.

    (strstr (clientline, " HTTP"))[0] = 0;

    // print the file we want

    Serial.println(filename);

    if (! file.open(&root, filename, O_READ)) {

        client.println("HTTP/1.1 404 Not Found");

        client.println("Content-Type: text/html");

        client.println();

        client.println("<h2>File Not Found!</h2>");

        break;

    }

    Serial.println("Opened!");

    client.println("HTTP/1.1 200 OK");

    client.println("Content-Type: text/plain");

    client.println();

```

```

    int16_t c;

    while ((c = file.read()) > 0) {

        // uncomment the serial to debug (slow!)

        //Serial.print((char)c);

        client.print((char)c);

    }

    file.close();
} else {

    // everything else is a 404

    client.println("HTTP/1.1 404 Not Found");
    client.println("Content-Type: text/html");
    client.println();
    client.println("<h2>File Not Found!</h2>");

}

break;

}

}

// give the web browser time to receive the data
delay(1);

client.stop();

}

}

```

Lets look at the new file listing code as well, its very similar, but now we've added a bit of HTML so that each file is part of a ****list and the name is a URL link.

```

void ListFiles(Client client, uint8_t flags) {

    // This code is just copied from SdFile.cpp in the SDFat Library

```



```

// and tweaked to print to the client output in html!

dir_t p;

root.rewind ();

client.println("<ul>");

while (root.readDir(p) > 0) {

    // done if past last used entry

    if (p.name[0] == DIR_NAME_FREE) break;

    // skip deleted entry and entries for . and ..

    if (p.name[0] == DIR_NAME_DELETED || p.name[0] == '.') continue;

    // only list subdirectories and files

    if (!DIR_IS_FILE_OR_SUBDIR(&p)) continue;

    // print any indent spaces

    client.print("<li><a href=\"");

    for (uint8_t i = 0; i < 11; i++) {

        if (p.name[i] == ' ') continue;

        if (i == 8) {

            client.print('.');

        }

        client.print(p.name[i]);

    }

    client.print(">");

    // print file name with possible blank fill

    for (uint8_t i = 0; i < 11; i++) {

```

```
    if (p.name[i] == ' ') continue;

    if (i == 8) {
        client.print('.');
    }

    client.print(p.name[i]);
}

client.print("</a>");

if (DIR_IS_SUBDIR(&p)) {
    client.print('/');
}

// print modify date/time if requested
if (flags & LS_DATE) {
    root.printFatDate(p.lastWriteDate);
    client.print(' ');
    root.printFatTime(p.lastWriteTime);
}

// print size if requested
if (!DIR_IS_SUBDIR(&p) && (flags & LS_SIZE)) {
    client.print(' ');
    client.print(p.fileSize);
}

client.println("</li>");
}

client.println("</ul>");
```

```
}
```

OK upload the sketch already!

