

# SUBSYSTEMS

---

## Starship STEM Learning System

©2021 Subsystems



All program sketches can be found at <https://github.com/Subsystems-us/Starship>

# Contents

Introduction to the Starship.....	5
Introduction to Arduino .....	6
The Hardware.....	6
The Software .....	7
The Integrated Development Environment.....	8
Menu .....	9
File Menu .....	9
Edit Menu.....	10
Sketch Menu .....	10
Tools Menu .....	11
Help Menu .....	11
Quick Access Icons.....	12
Open Sketch Tabs .....	12
Code Area .....	12
Notification Area .....	12
System Log .....	12
Board Selected .....	12
Your First (Sample) Program.....	13
Serial Output .....	18
The Starship .....	20
Connecting the Starship .....	21
Turning on the Lights.....	23
Let's Talk Binary .....	23
The Bit .....	23
Converting from Binary.....	24
Binary and the Starship Lights.....	25
The Shift Register.....	25
Programming the Lights.....	30
Create a Subroutine .....	34

Am I on the Bridge? .....	38
Solar System Flyby .....	39
The FOR Loop .....	40
Let's get better at Binary.....	46
The Flowchart .....	46
Receiving Serial Data.....	50
The Completed Code.....	55
The Light Sensor.....	58
Remember the Atom .....	58
Voltage .....	59
Current.....	59
Resistance.....	60
Ohm's Law .....	60
A Better Light Display .....	65
The Temperature Sensor .....	67
The Color Sensor .....	71
Planet Scanner.....	74
Planet Descent and Landing.....	77
The Accelerometer.....	85
Collision Detection .....	88
What's next? .....	90
Appendix A.....	91
Starship_display .....	91
Starship_BridgeLights.....	94
Starship_planet_run.....	96
Starship_better_binary .....	100
Starship_light_sensor .....	104
Starship_color_test.....	107
Starship_lander.....	113
Starship_tilt .....	118
Starship_alarm.....	121

All program sketches can be found at <https://github.com/Subsystems-us/Starship>

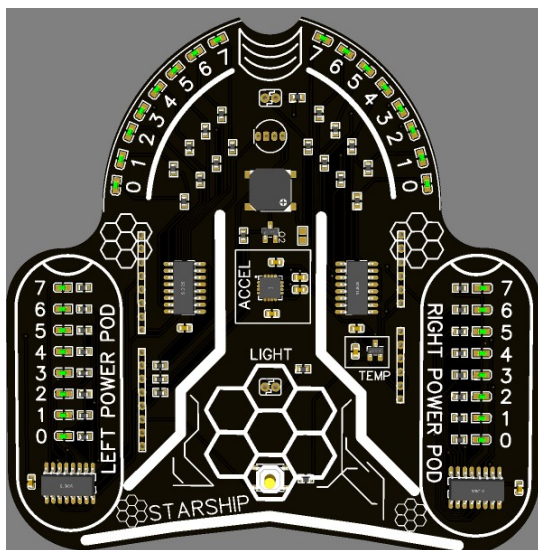
# Introduction to the Starship

The Subsystems Starship STEM Learning System (SSLS) is an Arduino shield that allows the exploration of programming, science, and engineering concepts. It has many elements, but all must be programmed to function. The shield includes the following components:

1. 4 separate 8 LED displays, accessed through a digital circuit called a shift register. This allows for interesting lighting effects and can be used to display information as binary numbers or other formats.
2. An ambient light sensor to measure the intensity of light in the area.
3. An ambient temperature sensor to measure temperature.
4. 3 LEDs (Red, Green, and Blue) coupled with a light sensor to be developed into a color sensing probe.
5. A pushbutton switch for allowing user input into programs.
6. An accelerometer to measure movement of the Starship.
7. A small speaker to generate sounds.

The Starship is an “Arduino shield.” Arduino is an open source hardware and software system to allow people to develop electronic devices easily. It has been used in the hobby community for years to create robots, 3D printers, and about anything else you can imagine. A shield is a separate circuit board that attaches to the basic Arduino board and takes advantage of the programming, processing, power supply, and communication available in the Arduino computer board.

To dive in to our STEM learning, we will first need to learn some basics about the Arduino computer and computer programming itself.



# Introduction to Arduino

Arduino (spoken Ar-dwee-no) is a computer and software company, user community, and open-source project that produce microcontroller kits and assembled units that connect the physical world to a small programmable computer. Arduino was established as a completely open-source project. That means the hardware and software designs are free to develop, produce, and distribute. This has made Arduino extremely popular in the hobby (frequently referred to as *Maker*) community.

The community has created a standard computer and interface, and has expanded it by adding software functionality using add on libraries. This means, if you want to add a display to your project, you will find many different kinds, all with a supporting library. So rather than having to write all code from scratch, you can import the library, hook up your display, and write simple commands like `print("Hello")` to display text. This means that you can get started on complex projects quickly.



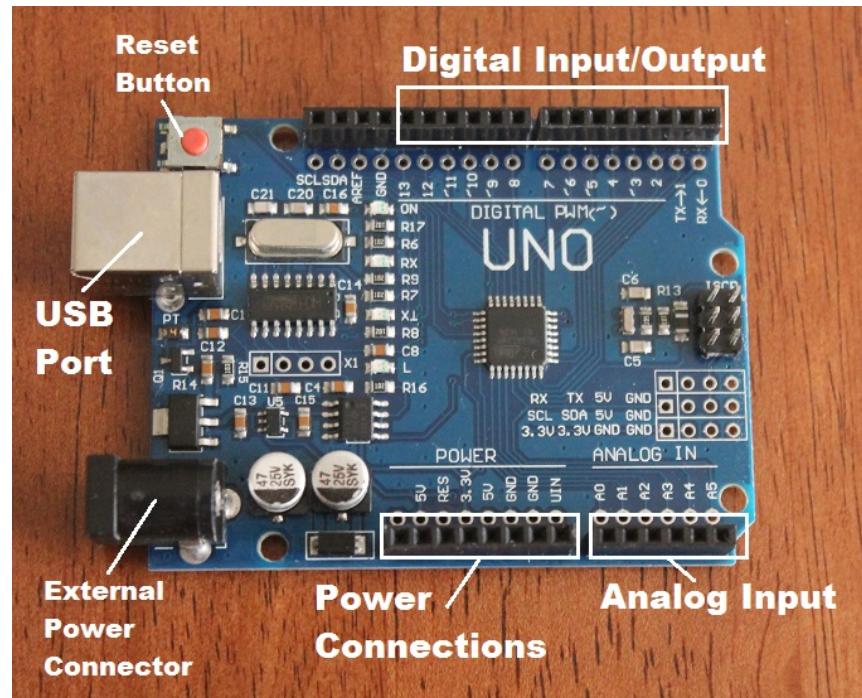
In addition to the base computer hardware, the Arduino community has produced many add on boards referred to as “shields.” These shields snap in to the pin headers on the computer board itself and route the computer input/output ports to various hardware on the shield.

Arduino is at the heart of many technologies used by hobbyists like mobile robots, 3-D printing, Internet of Things, and many others. As you start to play with this device, you will quickly realize that you are tinkering with an amazingly versatile and powerful device with an immense support community behind it and an internet full of easy weekend projects. It is also the heart of the Starship. The starship shield has no computer processor, power supply, or ability to communicate. We will take advantage of the Arduino for these things.

## The Hardware

The Arduino computer has a set of outputs that can create the signals needed to control many different electronic circuits as well as inputs that process data and make it available to our program.

A close up of the Arduino UNO board shows the connecting headers for the digital and analog input and output.



The UNO can be powered from the USB cable or from an external supply. We will be using the USB connection for both power and for uploading programs to the board.

Let's see how to connect to this computer so we can start programming.

## The Software

We will start our setup with loading the Arduino software and making sure it is configured properly. If you already have a current copy of Arduino on your computer, you can skip this section.

Visit [www.arduino.cc](http://www.arduino.cc) and download the latest version of the software for your particular system. Go to the main site and click on the menu item "Software" on the top navigation banner.

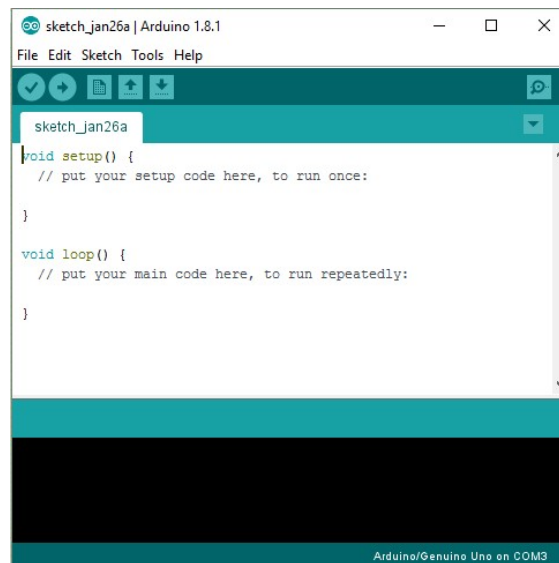
Scroll down until you find the link to download the IDE (integrated development environment). Arduino offers an online version of the IDE, but we will stick to the downloaded version for this training.

Click on the link for the Windows, iOS, or Linux version based on the computer you are using. Run the installation program. There is help on the Arduino site for installation if you run into problems.

Be sure to install needed drivers when prompted. These will allow your computer to recognize Arduino boards when plugged into your computer's USB port.

If all went well, the Arduino software and drivers are loaded on your computer. Let's open up that software and finish setting it up.

Find the Arduino program on your computer (from the Start icon in Windows or the Launchpad on a Mac) and run it. You should see the following:



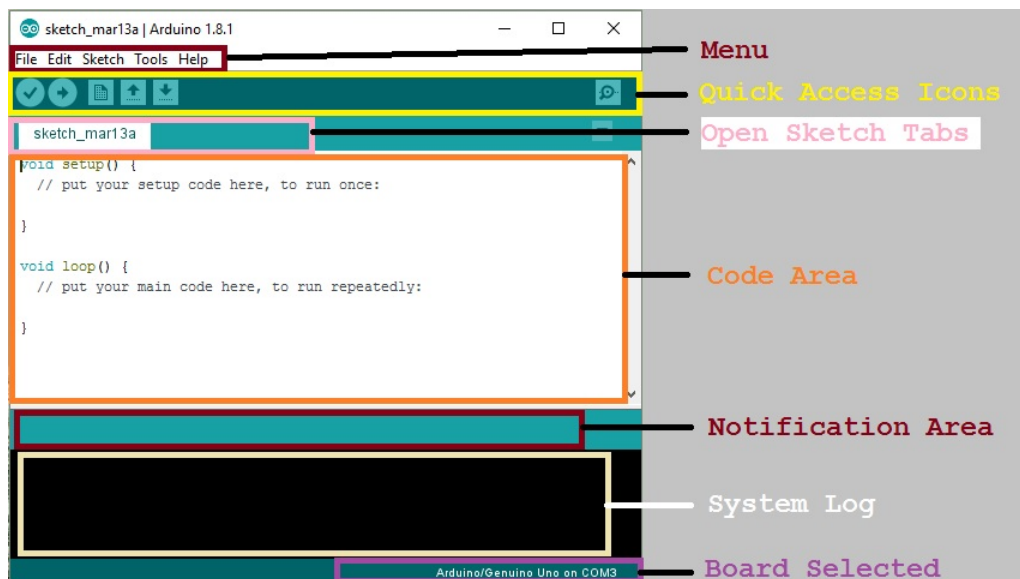
If you are presented with any other dialogs that show options for downloading other components, you can either accept them or cancel them. Your base software has everything we need.

## The Integrated Development Environment

Software development has changed since the dawn of computers. Initially, computers were programmed memory location by memory location using a series of switches and buttons. Then, more powerful computers and better hardware interfaces allowed us to program the computer using the computer. As computer complexity grew and more and more systems had computers embedded in them, computers were then used to write programs that were for other computer systems. This is called **cross-compiling**. Let's say you wanted to write an app that runs on an Android phone. But you want to write it on your Windows based computer. You could write the code in a text editor, and then send it to a small computer program (called a compiler) to convert it to the Android system language. The resulting file couldn't be run on your Windows based computer, but could be transferred to the Android phone and then run. This also opens us up to the ability to reuse code. There are some computer programming languages (like Java) that can be written once, and then compiled for many different platforms. The compiler does the work of taking the human readable program and converting it to the machine specific code to run on that platform.



But one of the biggest needs in programming is the ability to find problems with your completed software. This is called troubleshooting (or de-bugging). Wouldn't it be great if there were a way to test the program out on the machine you were writing the code on? Well, you can. A small program called an **emulator** can act like the device you are developing for and run your program locally. That means you can write your program, compile it, and run it on the Windows laptop, even though it is going to be loaded on an Android phone. Coordinating all this interaction, code de-bugging, and program monitoring is a piece of software called the **Integrated Development Environment (IDE)**. An IDE dramatically simplifies the task of writing computer software. Arduino is programmed in an IDE. You can use your Windows, Apple, or Linux computer to write programs and then compile them for Arduino. You can use the same software to download the compiled programs to the Arduino hardware and can even use it to monitor the hardware while it is running. Let's take a tour of the Arduino IDE.



Above shows a breakdown of the different areas of the IDE. Let's look at each area.

## Menu

The menu is the standard place for you to access all the functionality of the program. Note: Arduino refers to a program written in the IDE as a **sketch**.

## File Menu

**New** – creates a new Arduino sketch.

**Open...** - allows you to open a saved sketch.

**Open Recent** – this gives a convenient listing of the most recent sketches for you to select.

**Sketchbook** – This is a listing of all the sketches you have already saved.

**Examples** – this gives you access to the examples that come with the Arduino software as well as others that come with additional libraries you load.

**Close** – this closes the current sketch. It will prompt you to save if you have not.

**Save** – this will save the current sketch under the previously saved name. If it is a new sketch, you will be prompted to give it a name.

**Save As** – this allows you to save the current sketch as a new name.

**Page Setup** – this allows you to change the page characteristics. This is important if you will be printing your sketch.

**Print** – allows you to print the current sketch.

**Preferences** – this open a dialog that allows you to modify preferences and other setup related information.

**Quit** – this exits the program. You will be prompted to save your work if you have not.

## Edit Menu

The Edit menu selection has all the normal editing functions like cut, copy, paste, undo, etc. Of note are the two copy options (**Copy for Forum**, **Copy as HTML**). Since Arduino has such a large following, you will find many times you may want to post your code to a forum or on a webpage. These two options automatically format the code so it will have a standard appearance when pasted into these destinations.

## Sketch Menu

**Verify/Compile** – this selection compiles your sketch into a form ready for download to your Arduino hardware. You will see progress in the Notification Area and the System Log area. This is also where you will see if the compile was successful or if there were errors.

**Upload** – this will take a compiled sketch and upload it to connected Arduino hardware. If the software needs to compile the program first, it will do that.

**Upload Using Programmer** – this will upload a sketch to the Arduino hardware and overwrite the onboard software that communicates with the IDE. This is an advanced option for programmers who need to use the entire memory of the Arduino and will not need to communicate with the IDE anymore. DO NOT use this option to download programs.

**Export compiled Binary** – this will save the compiled program so you can have a local copy. This is useful if you want to back up the code or want to use another programmer to load it onto an Arduino.

**Show Sketch Folder** – this opens the file explorer on the folder where the current sketch is located.

**Include Library** – this allows you to select a library to add to your code. It will automatically add the correct “include” statement.

**Add File...** – this adds an existing file to the current sketch and opens it in another tab.

## Tools Menu

**Auto Format** – this puts your code in a standard, easy to read, format.

**Archive Sketch** - saves a copy of the sketch in .zip format.

**Fix Encoding & Reload** - fixes problems between the editor and other operating systems character maps when you copy or load sketches from other sources.

**Serial Monitor** - opens the serial monitor window for the currently selected Port. This allows the Arduino board to receive and send data in this window. This may reset the board (it will for your UNO board).

**Board** – this allows you to select the current board. The compiler needs to know the type of board to assign correct values to things like pins, ports, memories, and clocks.

**Port** - this contains all the serial devices available on your machine. It should automatically refresh every time you open the top-level tools menu.

**Programmer** – use this if you are using some other programmer than the built in one that comes with Arduino (we will be using the built in one).

**Burn Bootloader** – a bootloader is a small program that lets the Arduino communicate with your computer. This is what allows us to program it directly from the IDE. If you are using a brand-new chip or if your chip gets corrupt, you may need to download the bootloader to the chip again. This selection allows you to do that.







## Help Menu

Here you will find a lot of useful resources. Many of these link back to the Arduino page for more information. The Find in Reference selection is context sensitive and will use your cursor to give you help based on the command present at the cursor.

## Quick Access Icons



These are readily available icons that represent frequently used menu items. They are as follows:

	<b>Verify/Compile</b>
	<b>Upload</b>
	<b>New Sketch</b>
	<b>Open</b>
	<b>Save</b>
	<b>Serial Monitor</b>

## Open Sketch Tabs

These are tabs that select each of the sketches you have loaded. For most applications, you will just have one tab for the current sketch you are working on. As you use Arduino more and more, you will find that there may be standard routines that all your applications are using. You could save them in a separate sketch and open them in your current sketch as a separate tab. This will give you access to all the enclosed software routines.

## Code Area

This is where you will compose your code. This area has some great editor features. It allows the normal cut and paste functions. But it also will color code the lines depending on recognized functions and commands. This can be helpful and makes the code more readable.

## Notification Area

This space is used to convey the status of certain operations like save, compile, and upload. It will also turn red when there is an error that requires attention.

## System Log

The IDE calls different compilers and other helper applications during compiling and uploading. This area is used to display the information returned from those applications. Sometimes there will be extra information on errors that occur. After compiling, this area will show the amount of memory used and the percent of available memory for the particular device.

## Board Selected

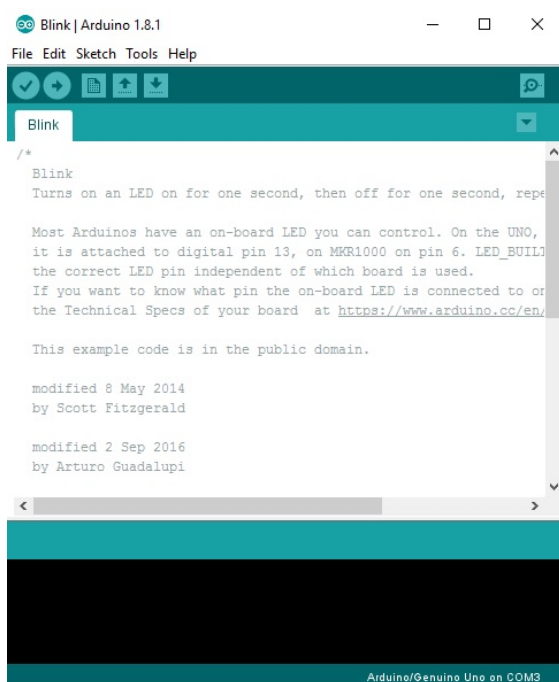
This area shows the currently selected board (set by Tools->Board:). This should match the board you are working with. If not, select it from the menu.

## Your First (Sample) Program

Now that you understand the basic layout of the IDE, let's start using some of the rich set of features it provides.

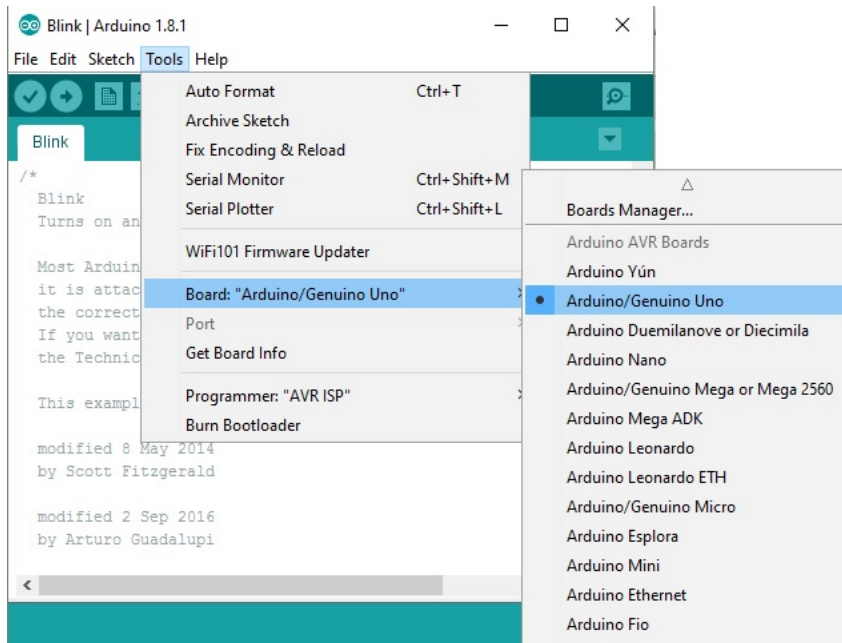
The easiest way to get started is with one of the included sample programs. From the menu bar select:

**File->Examples->01.Basics->Blink**



When you open the sketch, the code will appear in the code window. The Blink example will just load the code to blink the LED on the Arduino board. Let's hook up the board and get it ready to receive code. First, let's make sure that the IDE is set to the correct board. The board we will use is an Arduino UNO. From the menu, select:

**Tools->Board:->Arduino/Genuino UNO**

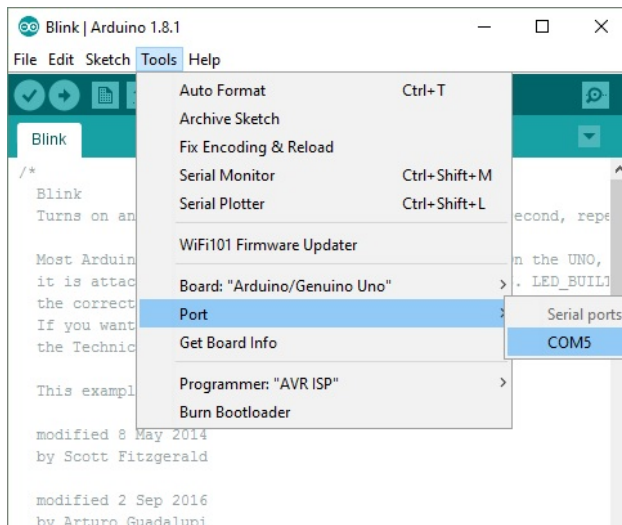


You can see from the board menu that Arduino has many variants. As you experiment with this computer system, you may find that want more inputs and outputs or you want to have a very small computer board that you can wear. Arduino and many other suppliers make many different boards that fill a variety of needs. But they can all be programmed with the Arduino IDE.

Plug the Arduino UNO into a USB port on your computer using the enclosed USB cable. The power light on the UNO will come on and your computer will register the new device. Now, let's tell the IDE what port our UNO is plugged into.

Select **Tools->Port->COM...**

When you view the selections, there should be a port labeled COM followed by a number. This is the reference to a serial port. Serial communication is when you send data one bit at a time over the transmission line.



In the above example, the UNO shows up on COM5. Whatever COM port is present, select it. If there are multiple COM ports, you could either select the different ports and attempt to download the program (described below) until you find the right port. Or you could go into your computer's hardware section and see which COM port the UNO is hooked up to. For most systems, there will most likely only be one COM port.

With the IDE selected to the correct board and the communication ready, we can now upload the Blink program to our UNO.

Select **Sketch->Upload** or click the Upload icon.



You will see a message in the notification area showing that the upload has begun as well as a progress bar. After a short time, you will see that the upload is complete and if you look at your UNO, you will see the red LED flashing slowly. Congratulations. You have compiled and downloaded your first Arduino program.

Let's look at the program to start learning how to write code. The first thing you notice is a lot of comment text. Programmers often want to place descriptions and instruction and even copyright notices within the program. This is good practice. You insert comments by using a `/**` on the line right before the comment. This lets the program compiler know that this is not part of the computer executable code.

```
// the setup function runs once when you press reset or power the board

void setup() {

    // initialize digital pin LED_BUILTIN as an output.
    pinMode(LED_BUILTIN, OUTPUT);

}
```

```
// the loop function runs over and over again forever

void loop() {

    digitalWrite(LED_BUILTIN, HIGH); // turn the LED on (HIGH is the voltage
level)

    delay(1000);                      // wait for a second

    digitalWrite(LED_BUILTIN, LOW);  // turn the LED off by making the voltage
LOW

    delay(1000);                      // wait for a second
}
```

This first thing to observe is that all Arduino programs have two main functions.

**void setup()** – this is where you place code that you want to execute once when the program first starts. This is where you will place your initialization code and where you will setup your hardware and start certain processes.

**void loop()** – this is where you place code that will continually repeat. This is normally where the bulk of your coding takes place. Code is sequentially executed and when it gets to the end, it recycles back to start executing code from the top of this function again.

The term **void** in front of the function names is a return type definition. When functions are executed, they run a specific set of code and then return a result. This result may be the value of a light sensor, the result of a math equation or the name of a person in your phone directory. It is some data that is being returned to the original program. When we declare functions, we must tell the software what type of data is being returned. Sometimes we return an integer number, sometimes a character, sometimes a Boolean value (True/False). If the function will not return anything, we declare it as void. These two functions do not return anything so they are declared with the void keyword.

The first line in the setup() function is:

```
pinMode(LED_BUILTIN, OUTPUT);
```

**pinMode** is a function that allows us to tell the Arduino how we want to use a specific pin on the computer. The UNO has 13 general purpose input/output pins. Before we use them, we must define how we will use them. The choices are:

- **INPUT** – the pin will be used to send data TO the computer.
- **OUTPUT** – the pin will be used to send data FROM the computer.



- **INPUT\_PULLUP** – the pin will be used as an INPUT, and the internal resistor connected to the pin will be enabled. This is used sometimes to ensure a digital input is either high or low. Without the input resistor, the input port voltage might just wonder around if no input is present and give a non-reliable reading of the input. By using an internal resistor tied to the positive supply, the port will go high with no input present. This puts the port in a known condition when there is no input present.

The **pinMode** function takes 2 arguments. The first is the desired pin. In the sample program, this is **LED\_BUILTIN** which the software already knows is assigned to pin 13 on the UNO. The second is the actual mode (**OUTPUT** for the sample program). Since we use the pin to light an LED, we are sending the signal from the computer to the LED so we need to establish it as an **OUTPUT**.

That is the only code required for the setup of our program. Now let's look at the loop function.

```
digitalWrite(LED_BUILTIN, HIGH) ;
```

The first command that is executed in the loop is a digitalWrite command. This command sends a High or Low signal out on the desired pin. The function takes two arguments. The first is the desired pin (**LED\_BUILTIN** for our example). The second is the state we want the pin in (**HIGH** in this example). When you write a **HIGH** to the **LED\_BUILTIN** pin (pin 13 on UNO), the LED on the board turns on.

```
delay(1000) ;
```

The next command is a delay function. This function takes one argument which is the number of milliseconds to pause program execution. In our example, it is 1000 milliseconds or 1 sec. So far in the loop we have turned on the LED and waited 1 second.

```
digitalWrite(LED_BUILTIN, LOW) ;
```

The next command is another digitalWrite. This time, it sets the LED port to **LOW**. This turns off the LED.

```
delay(1000) ;
```

The last command in the loop is another 1 second delay.

So in all, this bit of code turns on the LED, waits 1 second, turns off the LED, and waits one second. This has caused our LED to flash. Now that we reach the end of the loop, the loop is executed again and the LED flashes again. This loop continued indefinitely so the LED keeps flashing.

This was a simple example, but it incorporated a lot of the elements essential for writing your own programs. You got the IDE loaded and configured. You opened a sketch and downloaded it to the UNO. And you know the basics of outputting a HIGH or LOW on a port. You also understand the structure of a sketch and how the setup function is executed once and the loop function just continually repeats.

## Serial Output

One of the most convenient aspects of the UNO is that the same port that we use to upload sketches can also be used to download data. This is a great way to pass status, text results, and sensor data back to your computer.

Let's write a quick program to demonstrate this function. In the IDE, from the file menu, open a new sketch.

In the setup() function, add the following:

```
Serial.begin(9600) ;  
delay(1000) ;  
Serial.println("Program Running") ;  
Serial.print(5*4+2) ;
```

The first line tells the computer that we are going to be using the serial port and starts it up at a speed of 9600 baud. This is a measure of how fast the serial port will transfer data. This speed can be slow or very fast. The key is that both the sender and receiver must be using the same rate. Otherwise, the data will be corrupted. We will ensure our computer will be ready to receive this data flow shortly.

The next line is a one second delay like we saw in the Blink program.

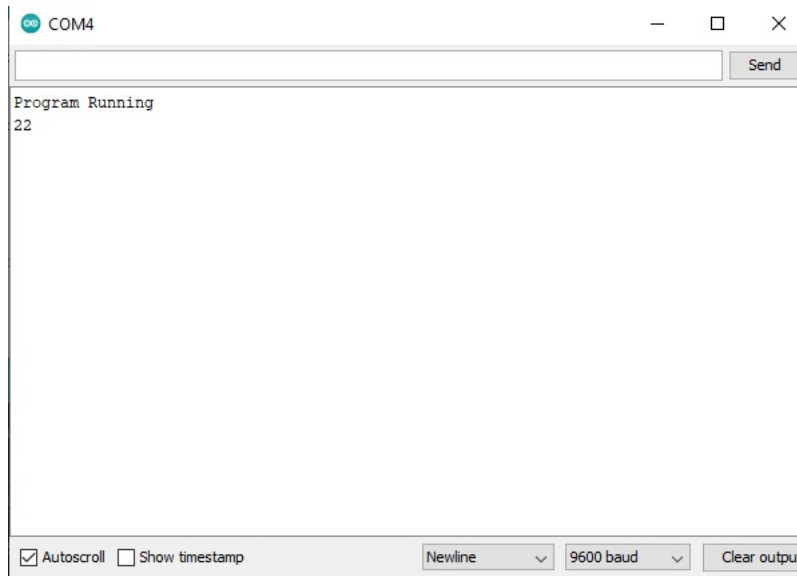
The next line accesses one of the built in functions of the Serial object we started. The function println() will print whatever is in the bracket to the Serial port. We can send any text this way but can also send numeric information like the last line in the program.

The difference between print() and println() is that println() will insert a newline directive after it prints so the cursor will go to the next line. This is helpful if you are displaying a list of items. If you use the print() function, the next thing printed to the Serial port will appear directly after the previous text.

Looking at our program, it should turn on the Serial port at 9600 baud, wait a second, display "Program Running," and then on the next line, Arduino will do the math on the formula we programmed ( $5*4+2$  which will equal 22) and print the result.

Download your program to the Arduino.

Now open the Serial Monitor. It is accessed from the Tools menu. When you open it (it will automatically connect to the port the Arduino is on), check the baud rate box in the lower right of the window. Make sure it is set to 9600. When you do, the program will rerun and you should see the output.

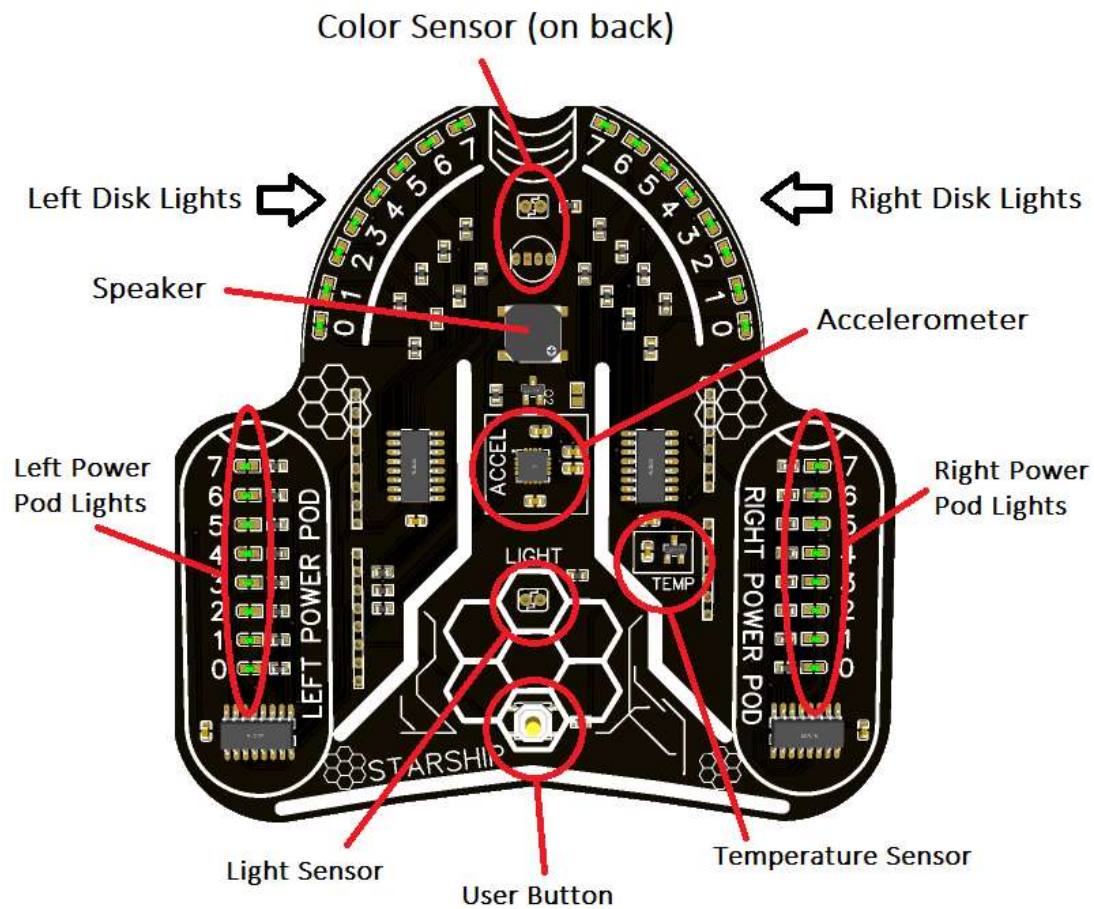


The Serial port is a great way to get easily readable data back from the Arduino and we will use it with our Starship.

With some of those basics out of the way, let's connect our Starship and start getting it ready to explore space!

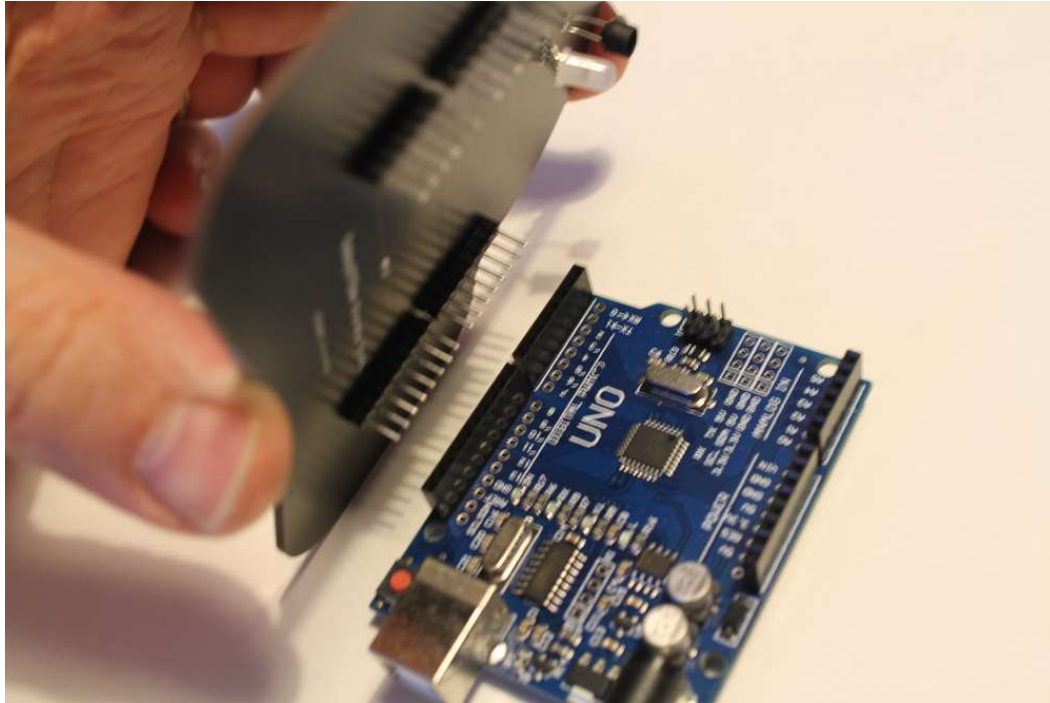
# The Starship

The Starship is a shield that will utilize the Arduino computer by providing input and output that we can program. Below is a diagram of the location of the main sensors and functions.

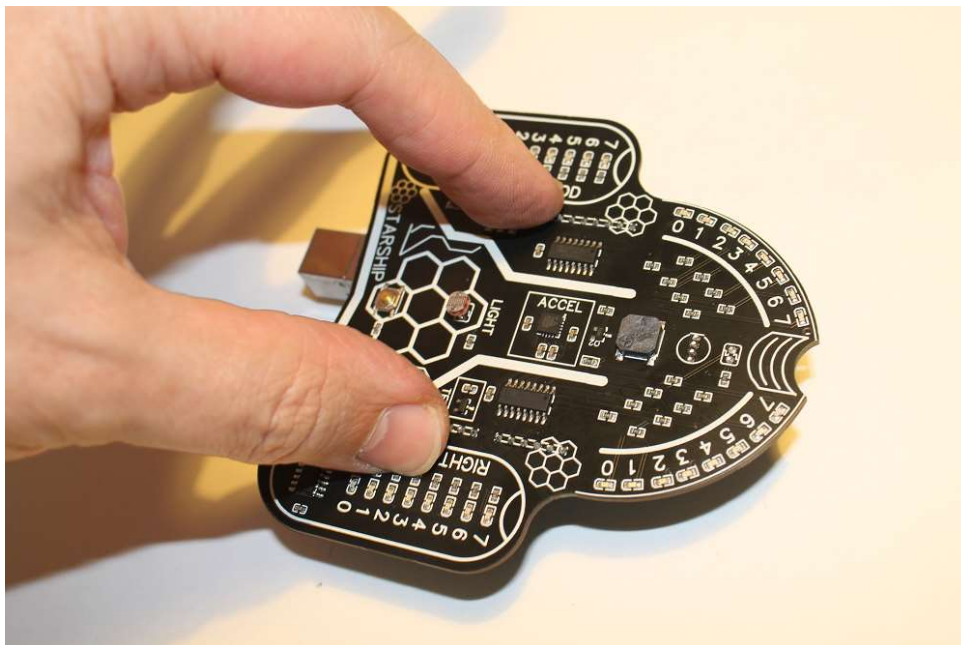


## Connecting the Starship

Start by connecting the Starship to the Arduino. Make sure the board is in the right orientation and the pins are lined up.



Line up the pins on the Starship with the connectors on the UNO. The gap in the pins matches the gap in the connectors. Gently press the shield onto the UNO until the pins are fully inserted.



When pressing the shield down into position, be sure to press near the pins on the main board to prevent putting force on the disk or energy pods.



One of the first things we want to tackle will be programming the shift registers so we can light the lights on the Starship.

**All of the programs in this curriculum are in Appendix A and at <https://github.com/Subsystems-us/Starship>**

Let's go.

# Turning on the Lights

There are a total of 32 lights on the Starship. If each light were individually controlled from a pin on Arduino, this would take 32 pins. The UNO only has 13 digital and 6 analog pins for a total of 19 pins. Plus, we want to do more than just light lights so some of these pins will have other functions. In addition, the Arduino computer can only supply so much power coming from the main computer chip. This would quickly over power the chip. We need to find some way to power the LEDs separately and be able to control all 32 LEDs with minimal pin usage on the Arduino.

There are a few ways to do this, but we are going to call into action a digital control chip called a Serial In / Parallel Out Shift Register. Before that, let's review binary numbers first.

## Let's Talk Binary

We communicate with words, sounds, and gestures. As sophisticated as computers are, they really do just communicate in ones and zeros. Computers are made up of millions of transistors and these transistors are like switches. They have two possible conditions; on or off. By combining multiple switches together, we can build a number system that the computer can then use to operate.

### The Bit

A **bit** is the smallest information that a computer can store. It is a single on/off state. So 1-bit can have two different possible states; 1 and 0. But let's look at a 2-bit system.

BIT 1	BIT 2	State Number
0	0	0
0	1	1
1	0	2
1	1	3

Since each bit can have two states, they can each be one or zero, so both together create 4 unique states (00, 01, 10, 11). So if we wanted to count to 3, we could use 2 bits to do it.

It is important to remember that counting systems start at zero so they always count up to one number less than the number of states. If you were asked to count in binary to 4, you would have

to include one more bit to do it (you would need 3 bits total). In fact, every bit we add will double the total number of states. There is actually a mathematical relationship between the number of bits and the number of states:

$$\text{Number of States} = \text{base}^{\text{Number of bits}}$$

So for a 4-bit data binary computer,

$$\text{Number of States} = 2^4 = 16$$

So, we can store numbers from 0 to 15 on a 4-bit computer.

## Converting from Binary

The way a binary number works is the same way our base 10 number system works. Look at the number 275 and how we represent it.

$$275 = 2 \times 100 + 7 \times 10 + 5 \times 1$$

We have all known from our earliest lessons on counting that there is a one's column, a 10's column, a 100's column, etc. Because we use a decimal system, all of our place holders are multiples of 10. All other base systems work the same way. Binary is a base 2 system so all of the placeholders will be multiples of 2. Let's look at what these placeholders look like.

Digit	Digit 5	Digit 4	Digit 3	Digit 2	Digit 1
Multiplier	16	8	4	2	1

Notice, just like base 10, as we move to the higher digits, we multiply the previous number by the base (2). Because this is base 2, we can only have a 0 or 1 in each digit location. So what is the decimal value of the binary number 1101? Solve it like the way we broke down the base 10 number above.

$$1101 = 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 = 13$$

So the number **1101** in binary is actually **13** in decimal. We can use this same procedure for converting from base 10 to binary. What is the decimal number 11 converted into binary?

Here we can look at the placeholder values in binary and see how to fit our number into its allowed values. From the chart you can see that the fifth digit represents a value of 16. This is too high for our number (11) so let's try the next smallest digit. Digit 4 represents a value of 8. There is definitely an 8 in 11 so we will start our conversion at that point showing a 1 in that



digit position. Since we have used up 8 of our initial 11, we are left with 3. The next digit represents a value of 4 which is bigger than our 3 so we will report no 4s. The next digit represents a value of 2. We definitely have a 2 in our 3 so we will report a 1 in this digit. Since we used 2 of our remaining 3, we only have 1 left. The last digit represents the 1's column so we can report the 1 we have left. The whole number is then constructed as follows:

$$11 = 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1$$

So if we pull off these 1s and 0s we get the binary number:

**1011**

This is the same as saying this number is 1 eight, and 1 two, and 1 one. Add them up and you get to 11. This is one method for converting between binary and decimal. Today, it is pretty easy to go online and find numerous calculators and apps that will do these conversions for you. It is good to understand number systems when dealing with computers because it is often necessary to represent numbers in different forms depending on how you want to process data.

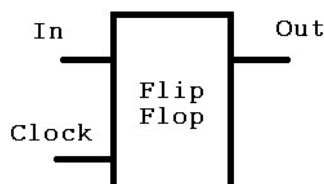
## Binary and the Starship Lights

Because the Starship has 4 sets of 8 LEDs, we could use an 8 digit binary number to represent which lights we want on (a 1 in that position) and which we want off (a 0 in that spot). We will now turn to a digital circuit that can take an 8 bit number and light LEDs with its outputs. That is a shift register.

## The Shift Register

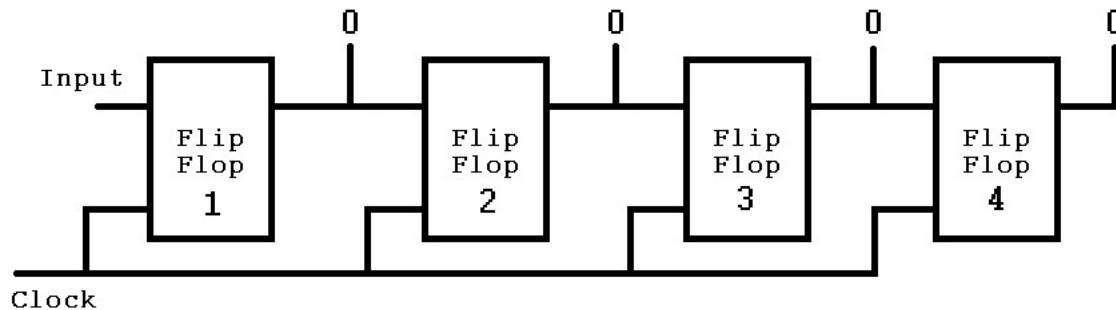
The shift register we will use takes in an 8-bit binary number one bit at a time and then transfers it to 8 output lines that each can handle enough current to light a LED. This is called a Serial In / Parallel Out shift register. The basic building block of this digital circuit is the Flip-Flop.

A flip-flop is a simple data holding circuit. There are many different types of flip-flops, but we will look at a simple version to understand how the shift register is constructed. A simplified block diagram is shown below.



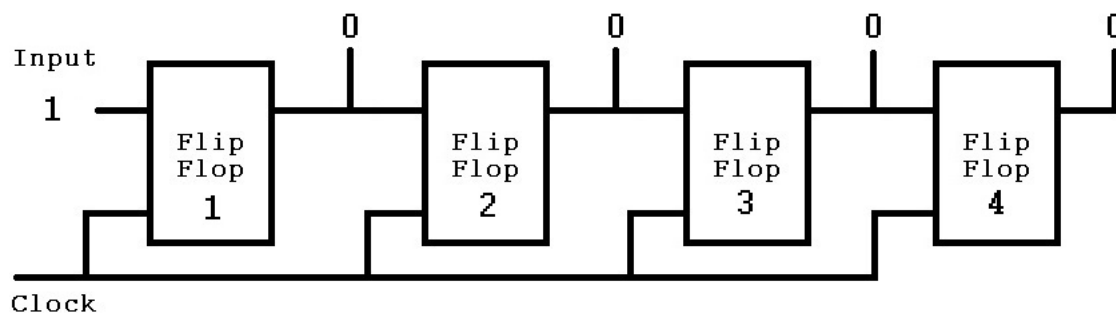
The flip-flop is a simple storage device that will transfer to the output whatever digital signal (1 or 0) we put on the input when we send a digital pulse to the clock input. That value will now stay on the output even if we change the value of the input. It will stay in this state until we send another clock pulse. At that time it will again transfer the digital signal from the input to the output. As you can see, it operates like a simple memory location, saving the data until we tell it to change.

Let's string a couple of these together to see how we can build a simple 4-bit shift register.



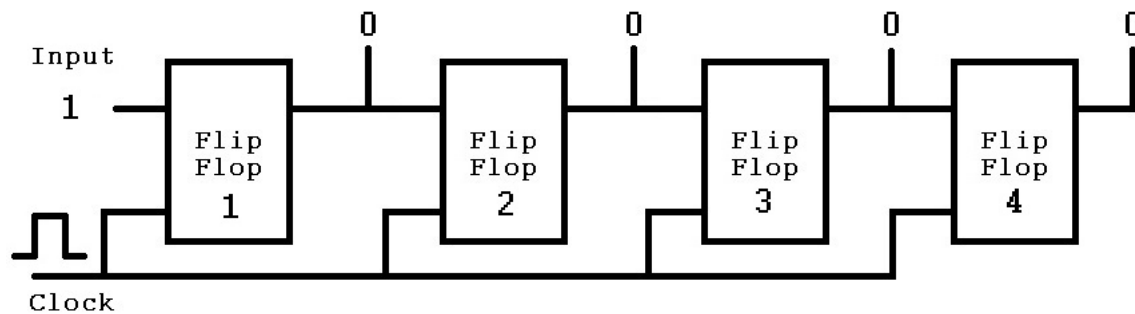
Here we have 4 flip flops. But you will notice that we tie each output of the first three to the input of the next flip flop. Also, all of the clock inputs are tied together. This is the basic layout of a shift register. With these two input lines (Input and Clock) we can shift in bits to each flip-flop.

Let's demonstrate this by shifting the binary number 1011. Remember the binary number discussion in the previous section. Using that information we can show that this is equivalent to a decimal 11. Let's see how we can recreate this 4-bit number with just two input lines.

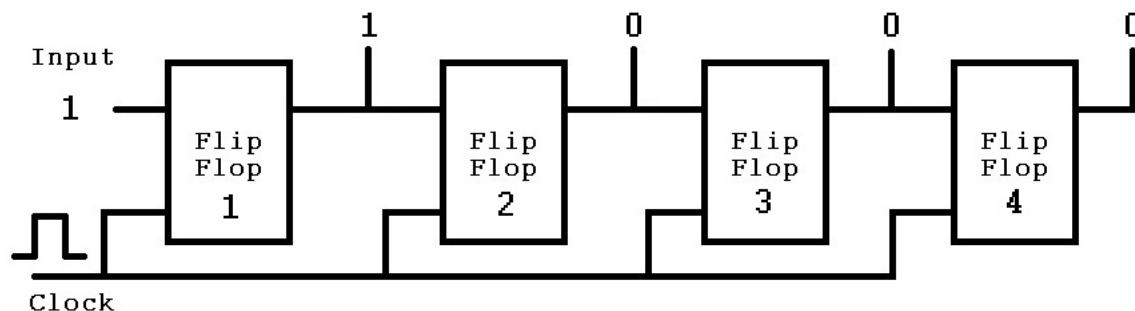


We will shift the number in from the Least Significant Bit (LSB) to the Most Significant Bit (MSB). The LSB is the digit to the most right in our number. It is the digit that has the least effect on the number when you change it. The right-most digit in binary represents the 1's column. Changing it changes the number value by 1. The MSB is the digit all the way to the left. The 4<sup>th</sup> digit over is the 8's column. It changes the value of a number by 8.

So the LSB of our number (1011) is 1. We applied this to the input in the above figure. Now, let's clock this value into the shift register.

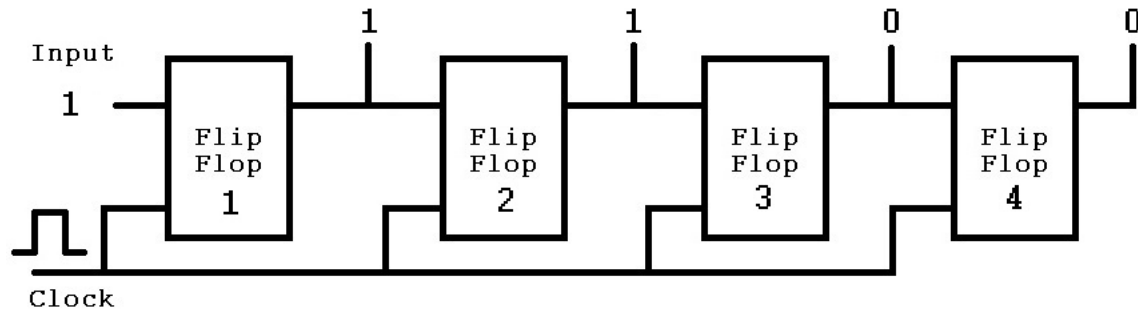


Here we show a clock pulse made up of a change from LOW to HIGH and then back to LOW again after a short time. These LOW and HIGH terms represent a digital value of False and True or 0 and 1 respectively. For our computer, it runs off of 5 Volts and will represent a digital False as near 0 Volts and a digital True of near 5 Volts. The clock pulse is applied to all of the 4 flip-flops so they will all transfer their input digital values to their outputs. This will leave us with the following state.

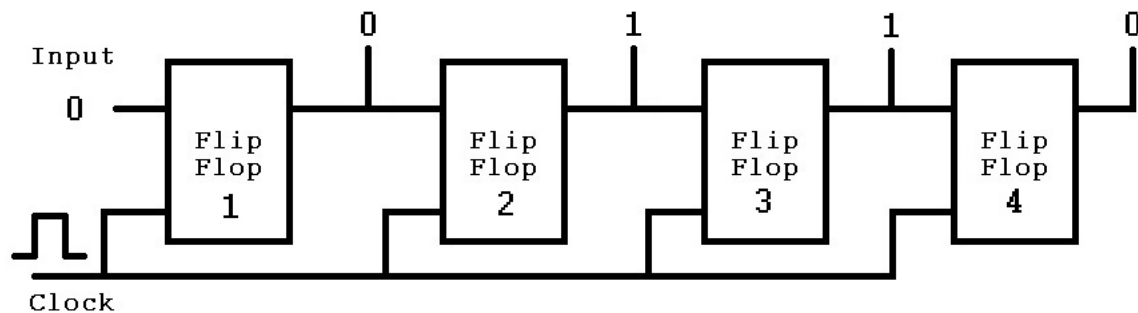


Flip Flop 1 had a 1 as the input and all others had a 0. After the transfer, Flip Flop 1 will have a 1 on the output and the others will have a zero. With the Clock LOW again, the transfer is disabled so we can change the inputs without affecting the outputs.

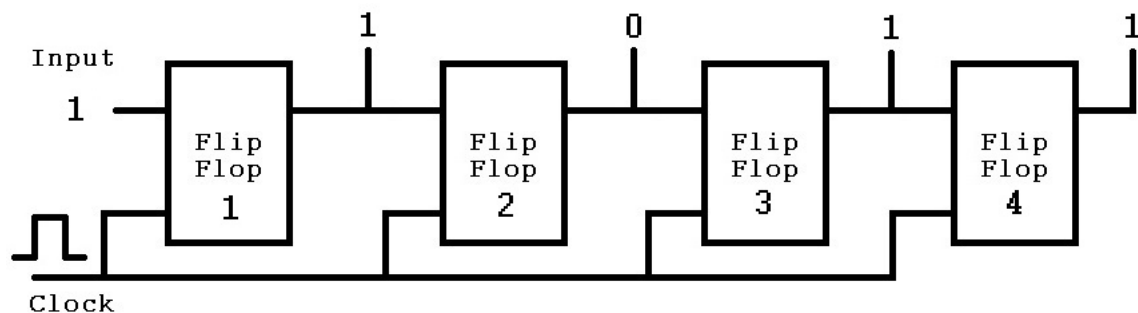
Let's shift in the next number. The second digit from the right in our number (1011) is a 1. We apply a 1 to the Input and then clock the register.



Flip Flops 1 and 2 had a 1 as an input so their outputs are now 1. Flip Flops 3 and 4 had a 0 input so they transfer a 0 to the outputs. The next number to shift in is a 0. We apply this to the input and clock the register.



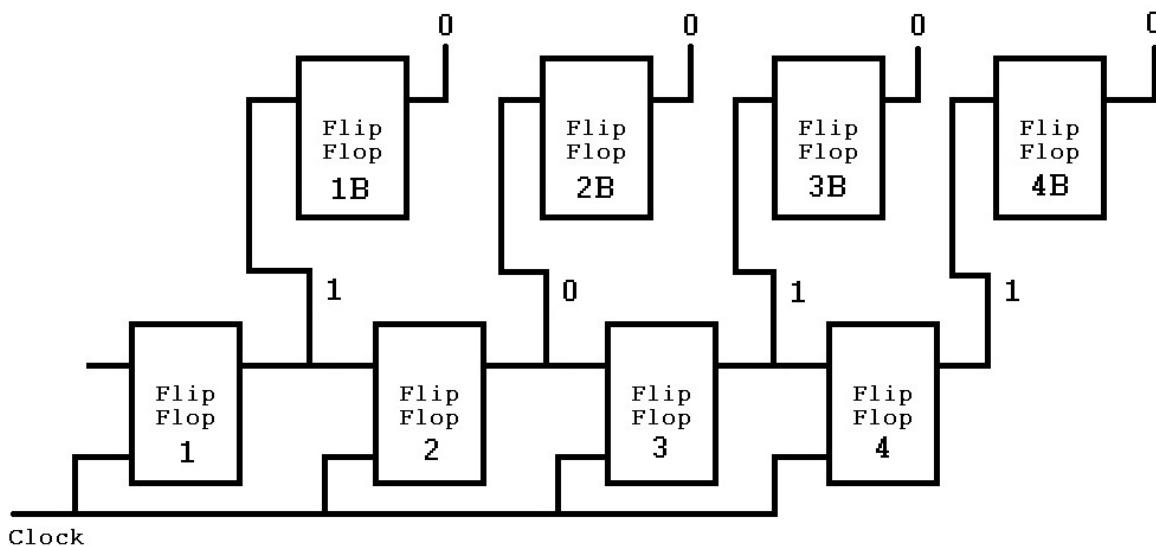
The Flip Flops with a 0 input now have a 0 output and the ones with a 1 input now have a 1 output. Let's shift in the final number. It is the MSB and is a 1. Placing a 1 on the Input and clocking the register gives the following.



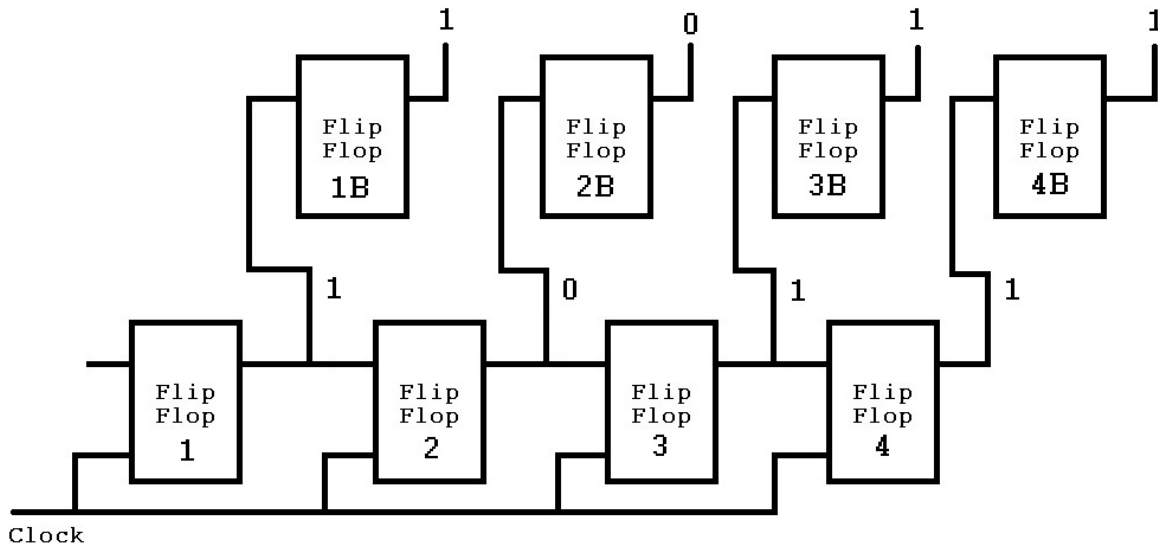
We are finished shifting in our number. Do you see it in the register? We started with the binary number 1011. If you collect the outputs of Flip Flops 1 through 4 you get 1011. The shift register now contains our number. We could continue to add Flip Flops to increase the number of bits we wanted to shift in.

Now, if you return to the computer outputs, every 1 in the output is really 5 Volts and every 0 is 0 Volts. If we connected lights to the outputs of the shift register, it would light each light that had 5 Volts and would not light the ones that had 0 Volts. This is exactly how we are going to control the lights on the Starship.

There is one other issue. If the outputs of this register were just hooked to the lights, we would not just see them come on but would watch the whole shifting process. This could cause issues with lights flickering. It would be great if, when we were done shifting the value into the register, we could then shift the final number to the lights. That would allow just the final shifted number to be displayed and not all of the shifting itself. Fortunately, we recall from the discussion of a flip-flop that it can be used to store data. If we add a flip-flop onto the output of each register flip-flop we could clock the register value onto these flip-flops and hold the number there until we want to change it. Let's do that.



The bottom section is just the shift register we have been discussing and Flip Flops 1B through 4B are the new “storage” flip-flops that are going to hold the final result of the shift register. The bottom shift register shows the final state of shifting in our number (1011). Notice that the B Flip Flops are showing all zeros. This is the value of them from a previous update. Now when we clock the B Flip Flops the bit values from the shift register will be transferred to the outputs.



Now the B registers hold the final value of the shift register. Now the shift register is free to start shifting another number without impacting the lights attached to the B Flip Flops.

You can see how useful these flip-flops are in digital circuitry.

So for our Starship, we have four separate 8-bit shift registers. The Clock and Input lines are all tied together. This means all four registers will shift the same values. But, each register will have its own storage register controlled by the computer. This way, we can transfer a number to all shift registers but only transfer the final number to the storage register we desire.

So counting the Input, Clock, and four Storage control lines, we will be able to independently control our 32 LEDs with 6 pins from the Arduino.

Let's write the code to do this.

## Programming the Lights

We already introduced the Arduino IDE. We have loaded, compiled, and downloaded some simple code to make a light blink. Now, we will get into more programming to allow us to control the Starship lights.

To make our programming task easier, we will start by defining some names that we will associate with the different connections to the Starship. Instead having to remember which pin on the Arduino is controlling which function, we will write a quick header in our code that will define the pins with particular names that are easier to remember and read in the code. The compiler (the software that converts our code to code the computer can run) has a directive

called `#define`. We can use the directive to use a name in place of a number or port designation. Open the Arduino IDE software, start a new sketch, and place the following as the first text in the file (before the `setup()` routine).

```
// Map the Arduino pins to the Starship functions
#define shift_data 3 // Data for the LED shift registers
#define shift_clk 4 // Clock for the LED shift registers
#define lpod_latch 5 // Register latch for the left Power Pod
#define rpod_latch 6 // Register latch for the right Power Pod
#define disk_left_latch 7 // Latch for the left disk LEDs
#define disk_right_latch 8 // Latch for the right disk LEDs
#define led_red 9 // Red LED for the color sensor
#define led_blue 10 // Blue LED for the color sensor
#define led_green 11 // Green LED for the color sensor
#define disk_speaker 12 // Speaker connection
#define button_input 13 // User button
#define temperature_sensor A0 // Temperature sensor input
#define light_sensor A1 // Light sensor input
#define color_sensor A2 // Color sensor input
#define accl_X A3 // Accelerometer analog X-axis signal
#define accl_Y A4 // Accelerometer analog Y-axis signal
#define accl_Z A5 // Accelerometer analog Z-axis signal
```

The `#define` keyword tells the compiler to replace the word following it to the associated number. For the first entry, when we use the term `shift_data` in our code and compile it to download to the computer, the compiler will replace the word with the number 3. The defines are as follows.

**shift\_data** – the common data (Input) lines to the shift registers  
**shift\_clk** – the common clock lines to the shift registers  
**lpod\_latch** – the latch to lock in the shifted data to the Left Power Pod  
**rpod\_latch** – the latch to lock in the shifted data to the Right Power Pod  
**disk\_left\_latch** – the latch to lock in the shifted data to the Left Disk Lights  
**disk\_right\_latch** – the latch to lock in the shifted data to the Right Disk Lights  
**led\_red** – power to the red LED used for color sensing  
**led\_blue** – power to the blue LED used for color sensing  
**led\_green** – power to the green LED used for color sensing  
**disk\_speaker** – the speaker on the Disk to play sounds  
**button\_input** – the pin the user button is attached to  
**temperature\_sensor** – the analog input pin connected to the temperature sensor  
**light\_sensor** – the analog input pin connected to the upper light sensor

**color\_sensor** – the analog input pin connected to the color sensing light sensor

**accl\_X** - the analog input pin connected to the accelerometer X-axis signal

**accl\_Y** - the analog input pin connected to the accelerometer Y-axis signal

**accl\_Z** - the analog input pin connected to the accelerometer Z-axis signal

By having this code in our program we can just use these names to reference the associated pins. If we were looking at our code and came across the line:

```
digitalWrite(10, HIGH);
```

It would not be obvious what this line is doing. We have seen the digitalWrite() command before in the LED Blink sample program, but what is the 10? This is a pin on Arduino, but what does setting it to HIGH do? If we use the #define keyword, the line now looks like this:

```
digitalWrite(led_blue, HIGH);
```

Now we can easily read that we want to set the blue LED to HIGH. This will turn it on. Having this code segment in all of our programs will streamline our development.

With this code inserted, we can tackle lighting the lights. Let's just start with the Left Power Pod.

We will now write the code to light the first and last lights on the Left Power Pod.

Since the shift register will need 8-bits of data, we want to send a number that has the first and last bit set to light the first and last lights. In binary this would be 1000001. When programming, we can represent numbers in different formats. The normal way to express numbers is in base 10, the normal decimal numbers that we normally use. But you can also tell the computer to use the number in base 8 (octal), base 2 (binary), or base 16 (hexadecimal). We tell the computer what type of base by the format of the number.

If we want decimal, we just use the number.	ex. 134
If we want octal, we start the number with a 0	ex. 0134
If we want hexadecimal, we start with 0x	ex. 0x35
If we want binary, we start with B	ex. B1000001

This is convenient if we know the number we want in a different base and don't want to convert it to decimal. We will do this in our program to make it easy for us too.

In the setup() section of your program, add the following:

```
pinMode(shift_data, OUTPUT);  
pinMode(shift_clk, OUTPUT);  
pinMode(lpod_latch , OUTPUT);
```



As before, this will setup the pins we need to do the desired function. In this case, all of the signals are from the computer to the shift register so they are all designated OUTPUT.

Now ,we need to lock in the output latch so it doesn't change while we are loading the data into the shift register. We do that by sending the pin lpod\_latch LOW. Add the following after the pinMode command.

```
digitalWrite(section, LOW);
```

Now we can shift our number into the shift register. Conveniently, there is a function that already exists to do this for us. Add the following line to the program.

```
shiftOut(shift_data, shift_clk, MSBFIRST, B10000001);
```

This is called a subroutine. It is a section of code that is called by the main program to do certain tasks and then return. You will remember that we were already introduced to the setup() and loop() subroutines. For those, we did not need to pass any information to them. For the shiftOut subroutine, we need to pass it information for it to work properly. Specifically, it need four pieces of data. The first is the pin we are using for the shift register data pin (shift\_data for us). The second is the shift register clock pin (shift\_clk for us). The next is a directive telling the subroutine whether we want to shift the number in forward or backward (MSB or LSB first). We want to shift MSB fist to line up our number with the output LEDs. The last is the value we want shifted out. That is the binary number 10000001 to light the first and last LEDs. The only thing remaining is to send the signal to set the latch HIGH so it will transfer the shifted in number to the output to light the LEDs.

```
digitalWrite(section, HIGH);
```

That should be all we need. We only want this to happen once so we loaded all the code in the setup() routine and we leave the loop() routine empty. The final program should look like this:

```
// Map the Arduino pins to the Starship functions  
#define shift_data 3 // Data for the LED shift registers  
#define shift_clk 4 // Clock for the LED shift registers  
#define lpod_latch 5 // Register latch for the left Power Pod  
#define rpod_latch 6 // Register latch for the right Power Pod  
#define disk_left_latch 7 // Latch for the left disk LEDs  
#define disk_right_latch 8 // Latch for the right disk LEDs  
#define led_red 9 // Red LED for the color sensor  
#define led_blue 10 // Blue LED for the color sensor  
#define led_green 11 // Green LED for the color sensor  
#define disk_speaker 12 // Speaker connection  
#define button_input 13 // User button
```

```

#define temperature_sensor A0 // Temperature sensor input
#define light_sensor A1 // Light sensor input
#define color_sensor A2 // Color sensor input
#define accl_X A3 // Accelerometer analog X-axis signal
#define accl_Y A4 // Accelerometer analog Y-axis signal
#define accl_Z A5 // Accelerometer analog Z-axis signal

void setup() {

    pinMode(shift_data, OUTPUT);
    pinMode(shift_clk, OUTPUT);
    pinMode(lpod_latch , OUTPUT);

    digitalWrite(lpod_latch, LOW);

    shiftOut(shift_data, shift_clk, MSBFIRST, B10000001);

    digitalWrite(lpod_latch, HIGH);
}

void loop() {
}

```

Download this program to the computer. The Left Pod Lights should show the first and last LEDs lit. The other lights on the Starship may be on or off randomly due to the shift registers not being initialized to all zeros. We will take care of that.

Now that we have the basic code to light lights on the left pod, let's write them into our own subroutine so we can use one piece of code to light the lights on any of the four sections.

## Create a Subroutine

As stated before, a subroutine is a segment of code that is called from the main program to perform tasks and return. We will write a subroutine to control the lights on the Starship. Like the shiftOut() subroutine in our code, our subroutine will take a number of arguments.

Arguments are data passed to the subroutine that it needs to perform its function. We will pass two arguments. The first will tell the subroutine which of the four shift registers we want to use and the second is the value we want it to display. Since these values are not known at programming time, we need to represent them with variables. A variable is a memory location that we declare to hold a piece of data. What we need to know to use it is the type of data we want to hold. Here are some data types used in the C programming language.

Type	Size	Values	Comment
CHAR	1 byte	0-255	An 8-bit number
INT	2 bytes	0 to 65535	16-bit integer number
FLOAT	4 bytes	2.3E-308 to 1.7E+308	Decimal number
DOUBLE	8 bytes	3.4E-4932 to 1.1E+4932	Large decimal number

A byte is 8 bits of data. Remember a bit is 1 or 0. Stringing 8 bits together gives you a byte. This is the basic unit of computing for the Arduino since it is an 8-bit computer. This means the computer internally communicates with 8 bits of data at a time. Each memory location holds 8 bits of data. Good programmers try to optimize their code to use the least amount of memory so knowing how much memory variable types take up is the key to limiting wasted memory.

For instance, if we had a variable in our program to hold the month someone was born, this value would only be from 1 to 12 so we could use 1 byte to store this. In this case, we could choose the CHAR data type because it is only 1 byte of memory. If we used DOUBLE instead, 7 bytes of this variable would never be used and would be wasted space in the computer's memory.

We can also define some of these variables as SIGNED and UNSIGNED. We would use UNSIGNED if we were not going to run into negative numbers. This maximizes the number we can get to (for INT, this is 255). If we have to allow for negative number, this is going to affect the size of the maximum number we can get to (for INT, it is -32768 to 32767).

For our subroutine, the variable that holds which shift register we want to use will only need to count to the highest pin number that we use (which is pin 8 for the Right Disk Lights) so we can use a CHAR type for that. Since the value we want to display is only 8-bits wide (one bit for each LED), we can use CHAR for the value argument as well.

Let's construct the subroutine with this data.

```
void dispNum(char ledSet, char ledValue){}
```

We start out by declaring this subroutine as void. This tells the software that it is not going to return any information. We get to choose our subroutine name. We want it to identify the function that it will perform to allow us to easily understand our code so we choose dispNum. Next, we know we are passing in two CHAR variables, but we haven't named them. Again, we want to use names that make the code easier to understand so we use ledSet to carry the number of the set of LEDs we want to change and ledValue for the value to change them to.

Now we can add the code we wrote before, but change the fixed values (lpod and B10000001) to these variables so we can use this subroutine generically.

```
void dispNum(char ledSet, char ledValue) {
    // set the desired latch low
    digitalWrite(ledSet, LOW);
    // shift out the desired number
    shiftOut(shift_data, shift_clk, MSBFIRST, ledValue);
    // set the desired latch high
    digitalWrite(ledSet, HIGH);
}
```

A subroutine is a standalone section of code, so we will place it outside of the setup() and loop() routines. Place the code after the loop() routine after the last bracket.

Looking at it, it is the same as the code we wrote for the Left Pod Lights, but now instead of just working with those lights, the register represented by ledSet is used and the value represented by ledValue is used.

To call this subroutine to energize the same lights we did before, we would just call:

```
dispNum(lpod_latch, B10000001);
```

Let's add this line to our code and show that we get the same results. The final code should look like this:

```
void setup() {
    pinMode(shift_data, OUTPUT);
    pinMode(shift_clk, OUTPUT);
    pinMode(lpod_latch, OUTPUT);
    dispNum(rpod_latch, B10000001);
}

void loop() {
}

void dispNum(char ledSet, char ledValue) {
    // set the desired latch low
    digitalWrite(ledSet, LOW);
    // shift out the desired number
    shiftOut(shift_data, shift_clk, MSBFIRST, ledValue);
    // set the desired latch high
    digitalWrite(ledSet, HIGH);
}
```

When you compile and download this code, the Left Pod Lights have the first and last lights lit. Again, you may notice that the other 3 sections of lights are randomly lit. Since we now have access to the other lights, let's not forget to setup their pins to outputs as well. In fact, just like we used the `#define` directive to help our programming, we should now setup all of the input and output pins. Add this code to your `setup()` routine:

```
// Set the pin Modes
pinMode(led_red, OUTPUT);
pinMode(led_blue, OUTPUT);
pinMode(led_green, OUTPUT);
pinMode(shift_data, OUTPUT);
pinMode(shift_clk, OUTPUT);
pinMode(lpod_latch , OUTPUT);
pinMode(rpod_latch , OUTPUT);
pinMode(disk_left_latch, OUTPUT);
pinMode(disk_right_latch, OUTPUT);
pinMode(disk_speaker, OUTPUT);
pinMode(button_input, INPUT);
```

We didn't need to include the analog inputs (light sensor, color sensor, accelerometer inputs) because the `analogRead()` instruction sets those to `INPUT` automatically. Now that we have everything defined, programming will be a snap. Let's write a quick subroutine to turn off all the lights when the Starship is first powered up.

```
void dispOff() {
  // Turn all lights off
  dispNum(rpod_latch, 0);
  dispNum(lpod_latch, 0);
  dispNum(disk_left_latch, 0);
  dispNum(disk_right_latch, 0);
}
```

Add a call to this subroutine to the `setup()` section to turn all the lights off initially upon energizing the Starship.

```
dispOff();
```

Now we have complete control of the lights. Let's have some fun.

## Am I on the Bridge?

One of the fun things to see in movies is the crazy lights that flicker around on the back control panels on spaceships. Let's program our lights to do this. It is actually pretty easy if we know a couple of things.

Our software can generate a random number for us. We use the function `random()`. By using `random(256)` the computer will generate a random number between 0 and 255. This is perfect for our 8-bit lights. We can then randomly set the Left Pod lights by writing:

```
dispNum(lpod_latch, random(256));
```

We can then add a short delay and do the next group and so on. The final code in the `loop()` routine could look like this:

```
void loop() {  
    dispNum(lpod_latch, random(256));  
    delay(250);  
    dispNum(rpod_latch, random(256));  
    delay(100);  
    dispNum(disk_right_latch, random(256));  
    delay(350);  
    dispNum(disk_left_latch, random(256));  
    delay(200);  
}
```

This will display a random number on the light group, wait a short time, and continue with the next group. When it is done, the `loop()` will keep repeating. When you download this you will see the random light display of a sci-fi spaceship bridge.

You have come a long way!! With a few commands and an understanding of a simple shift register you have taken command of the lights on your Starship. Next, let's use these lights to fly through the solar system.

# Solar System Flyby

Sometime it is hard to see the scale of the solar system. Usually we see pictures of the planets on the same page and we don't see the relationship of the distances. We are going to write a program that lets us experience what it would be like to fly from the sun to Neptune in a minute. This will help us understand how the planets are really distributed in the solar system.

First, let's do some math.

How fast would we be going if we were to go from the sun to Neptune in one minute?

The distance to Neptune is about 2.78 billion miles. If we were to cover this distance in one minute, the speed would be:

$$\frac{2780000000 \text{ mi}}{1 \text{ minute}} \times \frac{60 \text{ minute}}{\text{hr}} = \mathbf{1.67 \times 10^{11} \text{ mph}}$$

This is an amazingly big number. Let's see how many times the speed of light this value is.

$$\frac{1.67 \times 10^{11} \text{ mph}}{6.7 \times 10^8 \text{ mph}} = \text{approximately } \mathbf{250 \text{ times}}$$

This is a ridiculous speed. Almost 250 times the speed of light! We have quite a Starship.

Assuming we travel this speed and knowing that we show up at Neptune in one minute we can calculate the time of arrival for the rest of the planets. Let's use a ratio to do this. Remember that a ratio sets up a relationship that can then be related to other items to find a desired value. Since our speed is constant for the trip, we can use the ration of distance to time to find the values for the other planets.

$$\frac{60 \text{ seconds}}{\text{distance to Neptune}} = \frac{\text{time to planet}}{\text{planet distance}}$$

We know the distance to the planets so we can solve for the time to each planet. Let's set this ratio up to solve for Mercury:

$$\frac{60 \text{ seconds}}{2780000000 \text{ mi}} = \frac{\text{time to planet}}{36900000 \text{ mi}}$$

Solving for time to planet:

$$\text{time to planet} = \frac{60 \text{ seconds}}{2780000000 \text{ mi}} \times 36900000 \text{ mi} = \mathbf{0.796 \text{ seconds}}$$

So if it takes a minute to get to Neptune, we will arrive at Mercury before the first second has passed.

Let's do the calculation for the rest of the planets.

Planet	Distance (miles)	Time (seconds)
Mercury	36.9 million	0.796
Venus	67.4 million	1.45
Earth	92.9 million	2.01
Mars	142 million	3.06
Jupiter	474 million	10.2
Saturn	927 million	20.0
Uranus	1.84 billion	39.7
Neptune	2.78 billion	60.0

We want to create a visual display when the Starship passes each planet. Let's create an animation with the disk lights. We will cycle the lights from the front to the back of the disk. We already have a subroutine to light lights. We now just need to write the code to write them in the order we want.

To do this, we need code that will repeat 8 times, lighting the next light back each time. One method to do this is with the use of the for loop.

## The FOR Loop

The for loop is a looping structure that repeats a set of instructions until a looping condition is met and then exits the for loop.

The syntax for this command looks like the following:

```
for (int i = 0; i < 8; i=++) {}
```

In the brackets, you will find the information that controls the loop. The first term (int i = 0) establishes a variable (i) that will be used to keep track of the looping. The next term (i < 8) is the condition that the loop will use to determine if it will continue (if the condition is True) or if it will exit (the condition is False). In this case, the loop will continue until the condition i < 8 is no longer True. The last term is what to do with i after each time the loop repeats. In this case, we use the function ++ which means to add 1 to i each time the loop completes. It is the same as writing i=i+1, but it is quicker so programmers like that. The curly brackets then contain the code we want to keep executing. So the above code will create an integer variable i, set it to 0,



check if it is less than 8, run the code in the curly brackets, and then add 1 to i and start again with checking to see if i is less than 8, and so on. So this loop will cycle for i = 0, 1, 2, 3, 4, 5, 6, 7. On the next loop, i will equal 8 so the expression i < 8 will fail and the for loop will exit and the program will continue from after the for loop code. Let's use this to create a subroutine that will cycle the disk lights.

Our subroutine for lighting the lights uses a number passed to it in binary. We want the binary number to light the MSB first and then shift the bit right to display the next light down. Look at the following code that does this:

```
void diskSweep(long swpSpeed) {
    int num = B10000000;

    for (int i = 0; i < 9; i++) {
        dispNum(disk_left_latch, num);
        dispNum(disk_right_latch, num);
        num = num / 2;
        delay(swpSpeed);
    }
}
```

We start our subroutine as we did before. We choose a name that describes the function of the subroutine (diskSweep). We are just creating an animation that uses a given set of LEDs (disk\_left\_latch, disk\_right\_latch) so we don't need to pass that as information to the subroutine. We do want to create a subroutine that can vary the speed that the lights cycle so let's pass in a long integer with a delay in milliseconds so the user can vary how quickly the lights cycle. We call it swpSpeed.

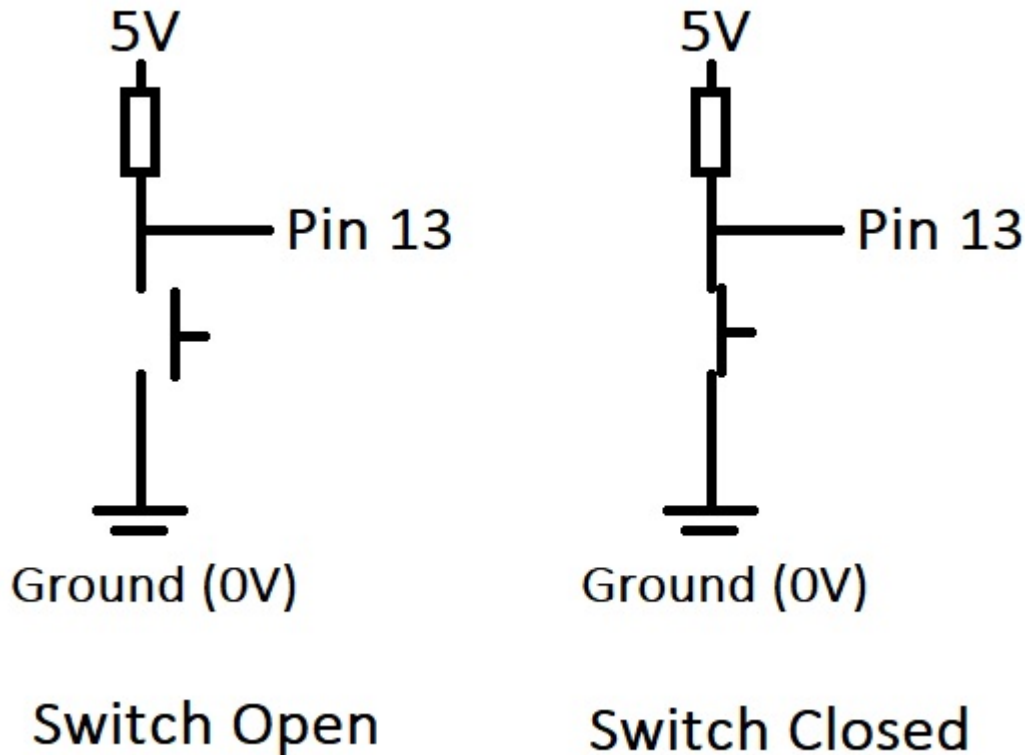
We then declare a variable named num and assign it the binary number with a 1 in the 8<sup>th</sup> position. When this gets sent to the dispNum routine it will light the front most lights on the disk. We then enter the for loop. In the loop we display the value of num on the left and right disk lights. We then divide num by 2 and assign the result back to the variable num. The reason for this is that dividing a binary number by 2 has the same effect as shifting the bits one position to the right. This will setup num to light the next light down when the loop runs again. We delay the amount of milliseconds in the variable swpSpeed. The loop then runs until it does 8 loops. At that point we return to the main program.

For each planet passed, we want to do the following:

- 1) Cycle the disk lights
- 2) Track which planet we are at on the Left and Right Pod lights
- 3) Printout the planet name to the Serial port

If we write that code and separate them by a delay to get to the next planet, we will have the bulk of the code written. The only other thing to add would be to initiate the launch with the on board pushbutton. Let's see how this button works.

Below is a diagram of the button circuit before and after the button is pushed.



The switch is connected to the Arduino pin 13. Before it is pressed, pin 13 feels the 5 Volts from the power supply. This is felt as a HIGH or 1 on the Arduino pin. When the switch is closed (pressed), pin 13 now feels the 0 Volts or LOW. By using the `digitalRead` function, we can see this LOW or HIGH in our program. The small box in the diagram is a resistor. It is an electronic component that helps limit current in a circuit. If we did not have it there, when we closed the switch, it would directly connect the 5V to the 0V and we would probably damage the computer or the power supply circuit. This is called a short circuit. The resistor acts as a load and limits the current that can pass through it which protects our computer from excessive current that could burn it out.

We can poll pin 13 in our program to determine when the user pushes the launch button. An easy way to do this is with a while loop. We already discussed a for loop that cycles for a given amount of times. A while loop cycles until a certain condition is reached. If the condition is never reached, the loop will cycle indefinitely. Use the below code:

```
while (digitalRead(button_input)) {}
```

We will execute the code in the curly brackets (we have nothing to execute) until the condition in the brackets is False. We have put the digitalRead command in the brackets. As shown above, when the switch is not pressed, pin 13 should show 1, HIGH, or TRUE (all used interchangeably in talking about digital computers and code). Since it is evaluated as TRUE, the while loop will continue executing its code (which is nothing in this case). When the user presses the button, pin 13 will then show 0, LOW, or FALSE (again, all meaning the same thing in computer-speak). Then, the while loop will be FALSE and the code will execute the while loop and continue with the next command. So this simple line of code will halt the program at this point and wait for the button to be pressed. When it is pressed, the program will continue.

Let's use the Serial output also to track the names of the planets as we pass. The code is shown below.

```
void setup() {  
  // Put in the usual #defines, setup code  
  
  Serial.println("Ready for launch ...");  
  
  while (digitalRead(button_input)) {}  
  Serial.println("Start...");  
  
  delay(796);  
  diskSweep(25);  
  dispNum(rpod_latch, 1);  
  dispNum(lpod_latch, 1);  
  Serial.println("Mercury");  
  tone(disk_speaker, 500, 250);  
  
  delay(454); //We subtract the time to diskSweep (654-200)ms  
  diskSweep(25); // 25ms x 8 leds is 200ms  
  dispNum(rpod_latch, 2);  
  dispNum(lpod_latch, 2);  
  Serial.println("Venus");  
  tone(disk_speaker, 550, 250);  
  
  delay(360);  
  diskSweep(25);
```

```

dispNum(rpod_latch, 4);
dispNum(lpod_latch, 4);
Serial.println("Earth");
tone(disk_speaker, 600, 250);

delay(850);
diskSweep(25);
dispNum(rpod_latch, 8);
dispNum(lpod_latch, 8);
Serial.println("Mars");
tone(disk_speaker, 650, 250);

delay(6940);
diskSweep(25);
dispNum(rpod_latch, 16);
dispNum(lpod_latch, 16);
Serial.println("Jupiter");
tone(disk_speaker, 700, 250);

delay(9600);
diskSweep(25);
dispNum(rpod_latch, 32);
dispNum(lpod_latch, 32);
Serial.println("Saturn");
tone(disk_speaker, 750, 250);

delay(19500);
diskSweep(25);
dispNum(rpod_latch, 64);
dispNum(lpod_latch, 64);
Serial.println("Uranus");
tone(disk_speaker, 800, 250);

delay(20100);
diskSweep(25);

```

```

    dispNum(rpod_latch, 128);
    dispNum(lpod_latch, 128);
    Serial.println("Neptune");
    tone(disk_speaker, 850, 250);
}

void loop() {
}

// Remember to include the display routines dispNum, dispOff,
// and diskSeep

```

The code starts out pretty familiarly. We #define our pin names, set the INPUT/OUTPUT modes, start the Serial port, and clear all the lights. We print that the ship is ready and then wait for the pushbutton. Once pressed, we print that the engines are engaged and we have started the trip. Each planet has a piece of code handling the ship passing. Here is the code segment for Venus:

```

delay(454); //We subtract the time to diskSweep (654-200)ms
diskSweep(25); // 25ms x 8 leds is 200ms
dispNum(rpod_latch, 2);
dispNum(lpod_latch, 2);
Serial.println("Venus");
tone(disk_speaker, 550, 250);

```

The first line is the delay from the last point reached. We reached Mercury at 0.796 seconds and we get to Venus at 1.45 seconds. If we subtract, we get the time from Mercury to Venus ( $1.45 - 0.796 = 0.654$  seconds). Remember that the delay function takes time in milliseconds. There are 1000 milliseconds in a second so a 0.654 second delay would be 654 milliseconds. When we arrive at the planet we simulate passing its gravitational influence by cycling the disk lights quickly. Even though this is quick (25ms per light), it takes 200 ms to complete so we will subtract 200 ms from each planet delay to account for this. So the final delay before Venus is  $654 - 200$  ms or 454 ms. We will do this for each planet after Mercury.

After the delay, we have reached Venus so we cycle the disk lights to show we arrived, light the second lights on the Left and Right pods, print out the planet name to the Serial port, and... oops. What is that tone command? This is a built in function to play a tone on a speaker attached to a pin. Our speaker is on pin 12 (we #defined as disk\_speaker). We send the tone function the pin,

the frequency we want played (550 Hz in this case), and the duration in milliseconds. As it gets further away the tone gets higher.

Download and run the code. Open the Serial Monitor so you can see the text printed out. When you are ready, press the button and launch the Starship. It is fun to see a time representation of the distance of the planets. Maybe you are a little surprised at how quickly the inner planets pass and how far away the outer planets are. Remember too that we are traveling at almost 250 times the speed of light.

The Voyager 1 spacecraft is currently about 14 billion miles from the sun. You could add another code section for it. You will just need to be patient as you wait about 5 minutes to pass it!

## Let's get better at Binary

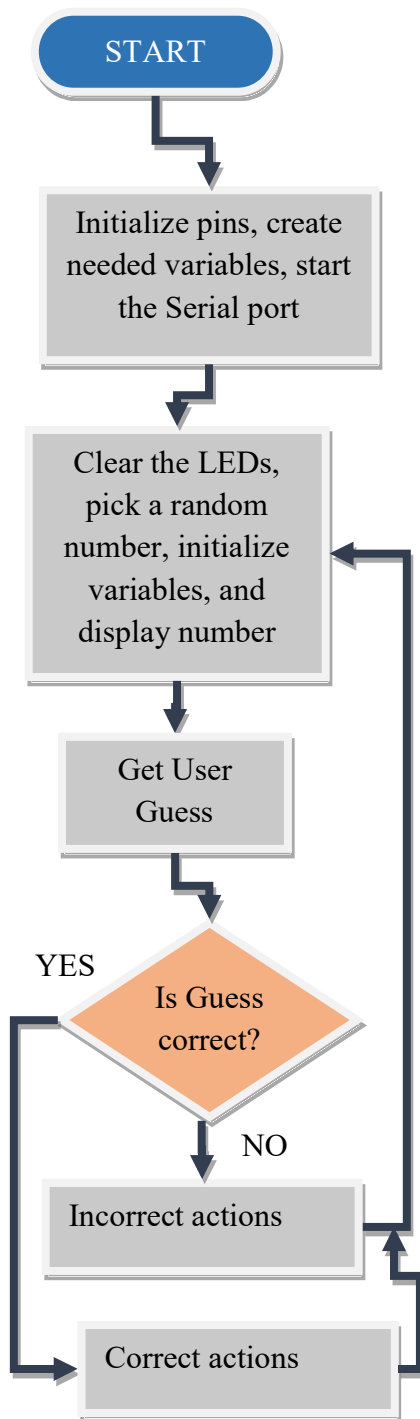
Our Starship computer is pretty sophisticated so let's have it quiz us on our binary number knowledge.

The Starship will generate a random number between 0 and 255 and display it on the Left Power Pod. You will then convert the number to base 10 (decimal) and send it to the Starship via the Serial port. It will display your guess on the Right Power Pod so you can compare the sequence. It will then display a light display if you are right or report via the Serial port that you were incorrect and show the answer.

We already have most of the programming knowledge to do this. But let's develop this program using a method that some programmers use to organize and layout how to write their code. We will use a flow chart.

## The Flowchart

Let's now shift to a time honored way of organizing our resources and concept into a logical sequence of events. Flow charts are great for this. A flow chart is just a graphical representation of the flow of logic and instructions in a computer program. It will be what we use to actually produce our code. Let's look at how it might be laid out.



Flowcharts often begin with a Terminator (a block that starts or stops a process). Let's use one that says "START" to begin our design.

This is a Process block. It is used for things that need to be done. We will use this block to setup our initial conditions. Let's make sure we have setup the pins, created the variables we need, and start up the Serial port so we can use it for input and output.

Here is where we start the main part of our program. We clear out LED displays, get a random number, initialize the variable values, display the random number, and wait for the user to guess.

We will need to write a routine that waits for user input and then receives it from the Serial port and returns the number that the user guessed.

Here is a decision point. We will evaluate whether the guess was right or wrong and take different actions accordingly.

This block represents the code we will run if the user guesses wrong. Let's light the Red LEDs on the Disk section and report the correct number via the Serial port. After that, we return to start another quiz.

This is the block we will use to represent what we will do if the user guess is right. Let's light the Green LEDs on the disk and return to the start for another quiz.

Let's use our flowchart to construct our program. We have broken the whole program down to simple blocks, so let's tackle each one separately and then combine them into the final program.

The Start block does not perform any functions so we will move to the next block.

Initialize pins, create  
needed variables, start  
the Serial port

We already know about the `#define` statements and that we want to initialize all of the Starship pins with the `pinMode()` statements. The other thing we need to do is define variables. Sometimes this part is done after the other blocks are complete when we know all of the variables that we need. The variables we define here are what are known as “Global” variables. They will be visible to all of the code in our program. They can be read and written to from the `setup()`, `loop()`, any subroutine and functions that we write. There are times when this is a good thing. However, there are dangers to this also. Some parts of code may alter the variable at a time when another part of code needed the current value. Because of the chance of corrupting data, programmers try to stay away from global variables. If you don't use them, you need to come up with other ways to pass this data to other code segments. This can make your code more complicated and require you to track which part has access and which does not. This concept of the ability to access a variable is called its “scope of visibility.”

We developed a subroutine to display the lights on our Starship. We passed in information to it in the parentheses that followed the subroutine name:

```
void dispNum(char ledSet, char ledValue) {}
```

This is the way we got information into the subroutine. If the `dispNum` subroutine tried to access a variable in the `loop()` routine, we would have gotten an error saying the variable does not exist. That is because variables created in the `loop()` routine are only available to the code in the `loop()` routine. These variables have what is called a “local” scope. When you define your variables outside of all of the routines, they now belong to the whole program and thus we call them “global” variables. This is very important to remember. Because of this, we will wait to define variables until we need them. At that point we can decide if they will be local or global.

The last thing we want in our setup has to do with generating random numbers. The computer cannot pick a purely random number. It uses an algorithm to generate them. This is why it is referred to as “pseudo-random.” This can cause a problem in our code. Every time we run our program it will generate the same sequence of random numbers. You could easily memorize (or get bored) with this. The computer does have something to help us out. We can seed this random



number generator. That is, we can tell the computer to jump to some random place in its sequence. I know, it sounds like we need another random number. Well, we kind of do. Luckily, there are a couple of options for us. One is to use the function `millis()`. This returns the amount of milliseconds that have elapsed since we powered on the Arduino. It is extremely probable that every time you run this program, you will not be at the exact number of milliseconds as the last. So this can introduce a randomness that could work for us. Fortunately, we have another random source. Our light sensor is always measuring the ambient light and converting it to a digital signal. We could use that for the seed to randomize our generator. That code will look like this:

```
randomSeed(analogRead(light_sensor));
```

We will feed the generator with this random signal coming from the light sensor. Now the program will not start generating the same “random” sequence every time we start it.

That looks like it for the first block of code. Let’s continue with the next block.

Clear the LEDs,  
pick a random  
number, initialize  
variables, and  
display number

Clearing the LEDs is easy. We already wrote code to do that.

```
// Turn all lights off  
dispOff();
```

To pick a random number, we need to declare a variable to hold it. We only need this number in the `loop()` routine so we will use a local variable:

```
int binaryNum = random(256);
```

Like before, this will generate a random number between 0 and 255. We already know how to display it:

```
// Display it on the Left Power Pod  
dispNum(lpod_latch, binaryNum);
```

We finished that segment of code. All looks good. Now we have to tackle the Serial input.

## Receiving Serial Data

This is not so straight forward as it would seem. The serial port sends data in the form of a code. For instance, how does the computer send the letter “A” to the user? A computer only deals with ones and zeros. The computer doesn’t even know what “A” is. Computer programmers have agreed upon a scheme where the numbers 0 through 127 (and actually a lot larger than that), will represent certain characters. This is called the American Standard Code for Information Interchange, or ASCII. If we send a ‘1’ to the computer over the Serial port, we are actually sending the number 49 in decimal. If you do an internet search for the ASCII table you will see the characters and the numbers that represent them in computer programming.

The command to retrieve Serial data is the `read()` command. It reads 1 byte of data from the Serial port at a time. In our program, the user could be sending a 1, 2, or 3 digit number (0-255) so we will need to handle this variable number of incoming digits. Luckily, our string of digits from the Serial port will have a very special character to end the data. This is the “new line” character. So all we need to do is to wait until the Serial port has data and retrieve it until we see the new line character. Then we will convert this 1, 2, or 3 character input to a decimal number. We will put this code in its own subroutine since it is a common task and we may want to use it in the future. Here is what it looks like:

```
byte getSerialData() {
```

Unlike the subroutines we wrote in the past, this code is going to return information back to the program. Because of that, we call it a function instead of a subroutine. We also need to declare it with the data type of the information that we are returning. In this case, it should be a number between 0 and 255 so the byte declaration is sufficient.

```
char receivedChars[4];    // an array to store the received data  
char charNum = 0; // Track which digit has been received  
boolean notDone = true; // Allow us to exit when the data is  
done  
char rc; // This will hold the character received
```

These are variable declarations for this function. **receivedChars[6]** is what is known as an array. You use an array when you want a series of variables to be able to be accessed by an index number. Think of the array as a street and the index as the house number on the street. You have

seen in the for loop that we can repeat a sequence of code and change the index every time the loop executes. This would allow us to walk down the street of our array and visit and perform the same function on every house. The [4] refers to how many house numbers I want to store in this array. We should only need a max of 3 for the digits we might have. But we will want at least 1 more so we can terminate this array with a given value (like we did with the Serial data) so we will know we are at the end.

**charNum** will hold the index of the current received digit from the Serial port. This will give us the place in the array to store the received data. We start at 0. The first element of the array is not 1, but 0. So `receivedChars[0]` is the data in the first element, `receivedChars[1]` is the second, and so on.

Since the computer deals in ones and zeros (or True and False), there should be a variable type that tracks that information. And there is. It is called Boolean and can hold a value of True (1) or False (0). **notDone** is our Boolean variable that will be True if we are still receiving Serial data and False when we are done. This will allow the software to look for this False value and exit the function.

**rc** is just the character that is received from the Serial port. Now that we have the variables we need (and all are local, nowhere else do they need to be used) we can continue.

```
while (Serial.available() == 0) {} // Wait until Serial data  
appears
```

`Serial.available` is a function of the `Serial` object. It returns whether there is data in the `Serial` receive buffer. The buffer is a set of memory locations where incoming data is stored before it is retrieved. Having a buffer allows you to continue to receive `Serial` data while the program is doing something else. This way you don't miss any data. When we call `read()`, we are actually not reading the current data at the `Serial` port but the data that is waiting to be retrieved in the buffer. This statement waits here until there is something in the buffer to retrieve. Once there is, we know the user sent it so we can start processing.

```
while (Serial.available() > 0 && notDone) { // while there is  
Serial data ...  
rc = Serial.read();
```

This line looks to see if data is in the buffer (`Serial.available() > 0`) and processes it. Notice the rest of the bracket (`&& notDone`). The `&&` are a type of Boolean math done on the two arguments. `&&` means AND. It says that in order to evaluate to TRUE, both `Serial.available()>0` AND `notDone` must both be True. We want to stop this loop getting data for two reasons. First, we run out of data (`Serial.available() = 0`) or we received the end of data character (`notDone = False`). That is to say I need both of these to be True for the loop to continue. If either one evaluates to False, the entire expression will evaluate to False and the while loop will stop looping. AND is a boolean math expression used a lot in computer programming.

We then get the current character from the Serial buffer.

```
delay(100); // This makes sure we don't miss data
```

Since the computer runs very fast and the Serial port was set to a low data rate (9600 baud) we want to delay a little to ensure, now that the data is being received, we give the transmission a chance to keep filling the buffer before we get back to that section of code. If we beat it there before the next character is received, we will have nothing in the buffer and end the while loop. This delay halts the program execution long enough to ensure we receive all the characters we need.

Now that we got a character from the Serial port, let's evaluate it. We can use an "if" statement for that. An if statement will execute a segment of code if the condition in the brackets evaluates to True.

```
if (rc != '\n') {  
    receivedChars[charNum] = rc;  
    charNum++;  
}
```

This segment looks for the "newline" character to see if we are finished receiving data. The exclamation point before the equal sign means NOT. We can use it before any Boolean expression to change a TRUE to FALSE, or a FALSE to TRUE. In this case the bracket reads "if rc is not equal to the "newline" character ..."

If it is not the newline, then it is valid data. The next line stores it in the array at location charNum. It then increments the charNum variable so it points to the next array location. The use of ++ is equivalent to charNum = charNum + 1.

```
else {  
    receivedChars[charNum] = '\0'; // terminate the string  
    notDone = false;  
}  
}
```

The "else" statement must follow an "if" statement and it executes code if the "if" statement evaluated to False. You can read this as "If True, do this. Else, do this." Here, we deal with the other case of the if statement (the newline character was sent). We terminate our array with the string termination character \0 so we know our array ends here. Strings are an array of characters normally used to display text. Again, the computer doesn't know what letters are so letters are stored as their ASCII values in an array and the text is terminated with \0 character to let the computer know the text is done. We also set the notDone variable to false because we are done and want to stop looking for more data. When we cycle back to the while loop, this false will

cause the combined expression (`Serial.available() > 0 && notDone`) to be false and the while loop will stop executing.

We now have an array of characters that represent our number, but we want the number itself. How do we change something that looks like `{'1','3','4'}` into the decimal value 134? There is a built in function that does that for us. it is called `atoi`. It takes an array of ASCII represented numbers and converts it to an integer number for us. This is the number that our function wants to return to the program. The following line converts the array to a number and returns it to the program.

```
return atoi(receivedChars);  
}
```

That is the whole function. Here it is in its entirety.

```
byte getSerialData() {  
    char receivedChars[4];    // an array to store the received data  
    char charNum = 0; //Track which digit has been received  
    boolean notDone = true; // Allow us to exit when the data is done  
    char rc; // This will hold the character received  
  
    while (Serial.available() == 0) {} // Wait until Serial data appears  
  
    while (Serial.available() > 0 && notDone) { // while there is Serial  
data ...  
        rc = Serial.read();  
  
        delay(100); // This makes sure we don't miss data  
  
        if (rc != '\n') {  
            receivedChars[charNum] = rc;  
            charNum++;  
        }  
        else {  
            receivedChars[charNum] = '\0'; // terminate the string  
            notDone = false;  
        }  
    }  
    return atoi(receivedChars);  
}
```

We have only given a brief description of these functions so you can understand them in the context of our program. You are encouraged to do an internet search on any of these topics to get more in depth information on these programming concepts.

Now that we have the function written, let's see how to call it from our program.

```
byte myGuess = getSerialData();
```

We define a local variable myGuess of type byte (because it only needs to hold a number between 0 and 255). We then set it equal to our function. Remember our function returns a number. This line of code will execute the function getSerialData() and assign the return value to myGuess. myGuess will now have the value inputted from the user through the Serial port.

Let's continue with our program.



Here, we can use our "if" statement to compare the guess with the actual number.

```
if (myGuess == binaryNum) {
```

Here, the if statement compares the values of myGuess and binaryNum to see if they are equal. Why the 2 sets of equal signs? In many programming languages a single equal sign is used for assignment only. That is, when we say Var1 = 0, we assign 0 to the variable Var1. The double equal sign is used to compare two values for equality. If we used a single equal sign in the expression, myGuess would be assigned the value of binaryNum and this is not what we want. We need the double equal sign to compare the values and return True if they are equal and False if they are not. This is a very common problem for beginning programmers so be careful with your equal signs.

If they are equal, we guessed right so let's light the Green Disk Lights (lights 1, 2, and 3) and send a Serial message.

```
// We were right!  
diskSweep(50);  
dispNum(disk_right_latch, B00000111); // Display Green Lights  
dispNum(disk_left_latch, B00000111);  
Serial.println("Correct!!");
```

The “else” statement will handle getting the answer wrong. We will display the red lights (lights 7 and 8), play a sad sound, and send a message with the correct answer.

```
} else {
  dispNum(disk_right_latch, B11000000); // Display Red Lights
  dispNum(disk_left_latch, B11000000);
  tone(disk_speaker, 500, 1000); // sound sad tone
  Serial.print("Sorry... the correct number was ");
  Serial.println(binaryNum);
}
```

Notice that we sent the first part of the message with the print statement instead of the println statement. This is so the printing cursor will stay at the end of the line and wait for the next print which will be the number. This formats all of this on the same line which makes it easy to read.

```
delay(3000); // Allow the user to see the results for 3 seconds
```

We then delay 3 seconds to allow the user time to see the results. This is the end of the run. At this point we can end the loop and cycle back for another run.

## The Completed Code

```
// Add all defines and routine setup code as before
```

```
void setup() {

  randomSeed(analogRead(light_sensor));

  // pinModes

  Serial.println("Ready to play ...");
}

void loop() {
  // Turn all lights off
  dispOff();

  int binaryNum = random(256);
```

```

// Display it on the Left Power Pod
dispNum(lpod_latch, binaryNum);

// Prompt the user for the guess
Serial.println("Enter your guess.");
byte myGuess = getSerialData();

// display the guess on the Right Pod
dispNum(rpod_latch, myGuess);
delay(2000);

// Did we get it right?
if (myGuess == binaryNum) {
  // We were right!
  diskSweep(50);
  dispNum(disk_right_latch, B00000111); // Display Green Lights
  dispNum(disk_left_latch, B00000111);
  Serial.println("Correct!!");
} else {
  dispNum(disk_right_latch, B11000000); // Display Red Lights
  dispNum(disk_left_latch, B11000000);
  tone(disk_speaker, 500, 1000); // sound sad tone
  Serial.print("Sorry... the correct number was ");
  Serial.println(binaryNum);
}
delay(3000); // Allow the user to see the results for 3 seconds
}

byte getSerialData() {
  char receivedChars[4]; // an array to store the received data
  char charNum = 0; //Track which digit has been received
  boolean notDone = true; // Allow us to exit when the data is done
  char rc; // This will hold the character received

  while (Serial.available() == 0) {} // Wait until Serial data appears

  while (Serial.available() > 0 && notDone) { // while there is Serial data ...
    rc = Serial.read();

    delay(100); // This makes sure we don't miss data
  }
}

```



```

if (rc != '\n') {
    receivedChars[charNum] = rc;
    charNum++;
}
else {
    receivedChars[charNum] = '\0'; // terminate the string
    notDone = false;
}
}
return atoi(receivedChars);
}

```

// Remember to add dispNum, dispOff, and diskSweep display routines.

Compile and download the code to the Starship. Open the Serial Monitor. Ensure you have Newline and 9600 baud selected.

When you see the text “Enter your guess,” look at the lights on the Left Power Pod. Convert the binary number displayed to decimal. As a review, here is the value of each light.

<b>Digit</b>	0	1	2	3	4	5	6	7
<b>Value</b>	1	2	4	8	16	32	64	128

For each light that is on, it is like a 1 in the associated column. Let’s look at converting the binary number 10011010.

<b>Digit</b>	1	0	0	1	1	0	1	0
<b>Value</b>	1	2	4	8	16	32	64	128

Now we just add up all of the numbers that have a 1 in the column.

$$1 + 8 + 16 + 64 = 89$$

So the decimal value is 89.

Keep doing runs with this on the Starship until you feel comfortable with these conversions.

Now that you have some awesome programming knowledge, let’s start to look at some of the sensors on the Starship.

# The Light Sensor

The light sensor is located on the main body of our Starship. It is actually a light sensitive resistor that changes its value depending on the amount of light that hits it.

We have already brought up the resistor when we discussed how it limits current in our switch circuit. Let's dive a little deeper into electronics and see how these little devices can be even more useful.

## Remember the Atom

If you remember from your study of matter, the Atom is the smallest particle of a substance that still retains all the properties of that substance. If I have a brick of gold, I can continually divide it until I get to one atom of gold. After that, splitting it in half will cause it to cease to be gold anymore.

The atom is comprised of a nucleus. The nucleus holds the subatomic particles protons and neutrons. Protons are positively charged particles and the number of them in the nucleus defines what material this atom forms. If you have one proton, you are Hydrogen. Two protons and you are Helium. The Periodic Table lists the known elements with their important parameters. The other particle is the Neutron. Neutrons have zero electric charge and their presence does not alter the element that the atom represents. However, neutrons affect the weight of the atom, and for bigger atoms, help hold the atom together. Since protons have a positive charge, they would tend to cause the nucleus to fly apart (like charges repel). But neutrons and protons have a very strong attractive force called the Strong Nuclear Force. This force operates over a short distance but helps overcome the electrostatic repulsive forces of the protons and holds the nucleus together. In electronics, we are not as concerned with the nucleus as we are with what is orbiting it.

Electrons are subatomic particles with a negative charge (equal to the protons positive charge) that orbit the central nucleus of an atom. In most atoms found naturally, there will be the same amount of protons and electrons so the atom will appear to have no charge since the positive proton charges will cancel the negative electron charges. Because electrons are free from the gluing effects of the strong nuclear force, they can actually be coaxed into leaving the bonds of their atom and move to another atom or just ejecting clear of the atom itself. Electrons of different materials have different abilities to leave their respective atoms. Metals tend to have very loosely bound outer electrons and they zip about between the atoms as if they can't decide which to belong to. This free movement of electrons is the basis of electrical current in a circuit and we even refer to these electrons as "free electrons." If a material has a large amount of free electrons, it is said to have high conductivity and is referred to as a conductor.

Other materials tend to hold on to their electrons and keep them pretty bound to the atom. In these materials, there are very few free electrons and the material does not conduct electricity well. We refer to these low conductivity materials as insulators.

If we take a material we call a conductor, like a piece of copper wire, we can inject electrons on one end and retrieve them on the other. The movement of the electrons through the wire is known as current, the force causing them to move is called voltage, and the small amount of difficulty the electron has moving through the wire is called resistance. Let's look at these more closely.

## Voltage

Voltage is defined as an electromotive force or difference of potential expressed in volts. It is a measure of this "pumping force" that causes electrons to flow in a circuit. It is important to recognize that voltage is always expressed as a difference. This means it is measured relative to some other point. If I told you that I was five years older, you would probably ask "older than what?" Telling you I am older doesn't mean anything until I give a reference. If I say I am five years older than my wife, you may still not know how old I am, but you do know the difference in age between my wife and me. Voltage works like that. You have probably seen a 9-volt battery. It sounds like it is giving you an absolute voltage by saying it is just 9 volts. But what it really means is that it has 9 volts of potential between its negative and positive connections. Often, we will tie the negative side of the battery or power supply to a "common" return line in our circuits. We arbitrarily establish this as a zero-volt point in our circuits and we measure all voltages with respect to it.

## Current

Current is a measure of the flow of electrons in a circuit. It is measured in Amperes or Amps for short. One ampere is a Coulomb number of electrons passing a point in one second. What is a Coulomb? Glad you asked.

**1 Coulomb =  $6.242 \times 10^{18}$  = 6,242,000,000,000,000 electrons**

That is a lot of electrons!! Because this number is so big, we often deal with current in milliamps (1/1000 of an Amp). For example, most computer older USB ports limit the supply of current to 500 mA.

## Resistance

Resistance is a measure of the opposition to current flow and is measured in Ohms. Resistance is anything that impedes the flow of electrons in our circuit. We will treat our conductors as though they have no resistance. But in reality, even the best conductors offer some resistance to current flow. A really thin wire may have around 1 Ohm of resistance for every meter. Most hookup wires, copper traces on printed circuit boards, components hookup leads, etc, have low enough resistance to assume a value of zero as we perform circuit calculations. However, if you run wires for long distances, you can no longer make this assumption.

## Ohm's Law

Ohm was a German schoolteacher and experimenter. He made the observation that if you connected a voltage source across a material, you would see a current running through the circuit. If you increase the voltage and keep the material the same, he noticed that the current increased. And if you kept the voltage constant and changed the material, the current would change. He would express this rule as a mathematical equation that would be referred to as Ohm's Law.

$$\text{Resistance } (R) = \frac{\text{Voltage } (E)}{\text{Current } (I)}$$

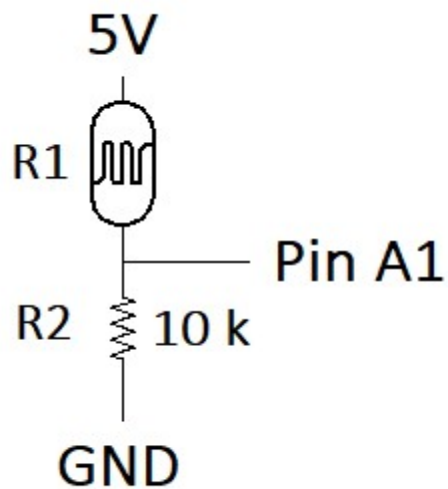
Using our symbols and rearranging the terms we get the following relationships:

$$R = \frac{E}{I}$$

$$I = \frac{E}{R}$$

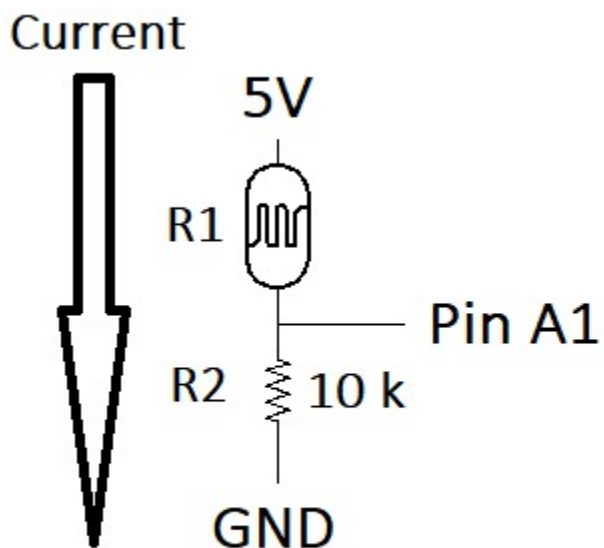
$$E = IR$$

These relationships are going to allow us to predict the behavior of circuits. For instance, let's look at temperature sensing circuit. When we showed the setup for our switch input, we showed a resistor in that line to limit current in the circuit. Here we will use a resistor in line with our light sensor to create a voltage divider circuit. Since our computer has an analog to digital converter, it can take an analog voltage (a voltage that can be any value between 0 V and 5 V) and convert it to a digital number we can use in our program. To create this varying voltage we will use the below circuit:



This circuit is hooked into the 5V supply from the computer. Current passes through our light sensor R1 (a variable value resistor), then a fixed value resistor R2 (10,000 Ohms), and then ends at the lowest point in our circuit (ground) which we establish as the 0 V reference point. We then tap off of the junction of the two resistors to get our varying voltage.

The current in the circuit will flow from the 5V supply to ground as shown below.



The current leaves the 5V supply, goes through resistor R1, then through resistor R2 and then to ground. The line connected to Pin A1 is just sensing the voltage at this point and can be assumed

to draw no current into it. That means the same current goes through R1 and R2. This is called a series circuit.

Let's calculate the current through the circuit using Ohms law.

$$I_{circuit} = \frac{E_{circuit}}{R_{circuit}}$$

The circuit voltage ( $E_{circuit}$ ) is 5V. The circuit resistance is the TOTAL circuit resistance. One of the common mistakes when using Ohms Law is applying it to the wrong values. That is a great reason to use subscripts for the terms. This will help prevent just using ANY resistance or ANY voltage in the calculation. In this case we need total circuit resistance. In a series circuit, resistors just add their values together. So the total resistance would be  $R1 + R2$  which equals  $R1 + 10k$ . Rewriting our equation with these values gives:

$$I_{circuit} = \frac{5V}{R1 + 10k}$$

If we know the value of R1, we can calculate the total circuit current. Since we have an equation for the circuit current, let's see how this relates to the voltage going to Pin A1 of our computer. Looking at our diagram, the voltage at Pin A1 measured to ground is the voltage over R2. Let's use Ohms Law to calculate this.

$$V_{R2} = I_{R2} \times R_{R2}$$

The current through R2 was calculated above and the resistance of R2 is 10k Ohms so filling in those values:

$$V_{R2} = \frac{5V}{R1 + 10k} \times 10k = 5V \left( \frac{10k}{R1 + 10k} \right)$$

So it turns out that the voltage felt across R2 (and sensed at Pin A1) is the supply voltage (5V) times a ratio of R2 to the total resistance. The resistance of R2 changes with temperature but let's assume that at room temperature it had a value of 10k. That means the voltage sensed at Pin A1 would be:

$$V_{R2} = 5V \left( \frac{10k}{R1 + 10k} \right) = 5V \left( \frac{10k}{10k + 10k} \right) = 5V \left( \frac{10}{20} \right) = 2.5V$$

This should make sense since two resistors of the same value with the same current going through them should have the same voltage across them. That means they will equally share the supply voltage and each would get 2.5V.

To measure this voltage on the computer, it uses an analog to digital converter. This takes the voltage on the pin and converts it to a digital number so we can use it in our program. The Starship (and its Arduino computer) has a 10 bit analog to digital converter. This will convert the 0 to 5V input to a number from 0 to 1023. This is the resolution of the converter. We can express it in terms of how many millivolts per digit it produces.

$$Resolution = \frac{5000mV}{1023} = 4.89 mV$$

So a voltage change of about 5 mV results in an increase of one on the converter result. This is important if we are engineering a system. If the sensor we are using changes only 5 mV through its entire range, the analog to digital converter could only move up by one number (like going from a reading of 500 to 501). This is not good enough resolution for that sensor and an engineer would need to select a sensor that would meet the resolution he required or add an amplifier to the sensor signal to put it more in the range of 0 to 5 V.

Let's take a look at the data coming from our light sensor.

Start a new Arduino sketch. Add the header information (the #define statements) and in the setup() routine, add the pinMode statements, start the Serial port and turn off all of the lights on the Starship. Add the display routines we came up with (dispNum, diskSweep). At this point, you can save this shell of a program since it is the starting point of just about every program you will write. Save it with a generic name like Starship\_Base so you will know this has all of the basic setup information needed. When you start a new program, you can open this Base sketch and then immediately use the Save As command in the file menu to save it as a different sketch. We could call this one Starship\_LightSensor. This will prevent having to retype this information every time.

Now that our sketch is setup, add the following to the loop() routine:

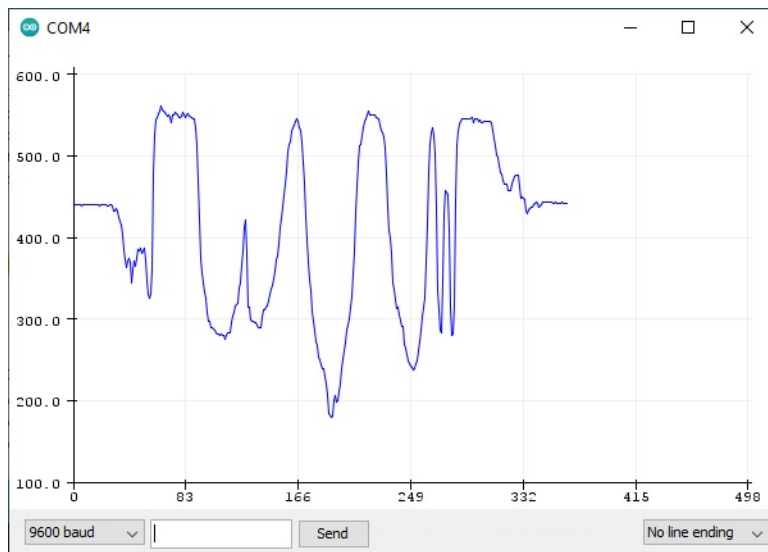
```
Serial.println(analogRead(light_sensor));  
delay(100);
```

This will send the analog reading on our light\_sensor pin to the Serial port so we can display it.

Compile and run the program.

Open the Serial Monitor and look at the output. You will see a sequence of numbers. The program reports the value every 100 msec. Move your hand over the light sensor and watch it change the received value. This is not the most convenient way to view this data. Close the Serial Monitor and open the Serial Plotter. It is the selection in the Tools menu just below the Serial Monitor.

Now the program displays a graph of the data. Wave your hand in front of the sensor again and watch the response on the graph. You should see the sensor respond similar to the graph below.



If you put the starship in bright light and then cover the sensor completely, you will have an idea of the range of values that come from this sensor. Close the Plotter and open the Monitor. You will see the numbers outputted by the program. Cover the light sensor and expose it to the brightness place in your room and note the minimum and maximum values. Write these values down. We will use them later.

Let's look at another way to view this data. Replace the `loop()` routine code with the following:

```
dispNum(lpod_latch, analogRead(light_sensor)/4);  
delay(100);
```

Download and run this code. Now the binary representation of the data will appear on the Left Power Pod. Did you see that we divided the analog result by 4? Why did we do this?

The analog to digital converter returns the value as a 10 bit number. Our power pods only display 8-bits of data. We want to preserve the most significant data which lies in the upper 8 bits. If a binary number is shifted 1 bit toward the LSB, it is the same as dividing by 2. For example, the binary number 100 = decimal 4. If we shift the number 1 bit to the right, we get 010 (decimal 2).



It has the same effect as dividing the number by 2. Since we need 2 shifts to turn the 10 bit number into an 8 bit number we divide by 2 and then divide by 2 again. This is the same as dividing by 4. That is why we did that. If we didn't, the power pod would display the lower 8 bits of the number and this would not account for the most significant information (the upper two bits). This shifting occurs so much in programming that there is a command that shifts for us.

If we want to shift right, we use `>>`. So if we had a variable called `myNum` and wanted to shift it 2 places to the right, we would use:

```
myNum >> 2
```

If you want to shift to the left, you would use `<<`. This is the same as multiplying by 2 for each shift.

Displaying the number is great if we wanted to know the exact value, but sometimes we want to display the intensity in a format that is more easily understandable. A bar graph would do this nicely. Let's develop the code to do this.

## **A Better Light Display**

Let's use the max and min values we got from our light sensor to create a bar graph display of the intensity.

We have a display routine to display a binary number, but we will only be displaying eight numbers. In binary, they would look like 00000000, 00000001, 00000011, 00000111, and so on. This will give us a growing display with light intensity. We can do this a few different ways, but let's start with the if statement we already know.

Look at the following code:

```
void loop() {  
  byte barData = B00000000;  
  int lightLevel = analogRead(light_sensor) / 4;  
  
  if (lightLevel > 126) barData = B11111111;  
  else if (lightLevel > 111) barData = B01111111;  
  else if (lightLevel > 97) barData = B00111111;  
  else if (lightLevel > 83) barData = B00011111;  
  else if (lightLevel > 68) barData = B00001111;  
  else if (lightLevel > 54) barData = B00000111;  
  else if (lightLevel > 39) barData = B00000011;  
}
```

```

    else if (lightLevel > 25) barData = B00000001;
    else barData = B00000000;

    dispNum(lpod_latch, barData);
    delay(100);
}

```

We start by declaring a variable `barData` of type byte. This will hold the final value that we send to the `dispNum` function. We then define a variable `lightLevel` of type integer (int). Note that we divide by four again to take the upper 8 bits of this 10 bit quantity.

When I got the values of minimum and maximum light intensity from the previous exercise, I got 25 and 140. If you want to display 8 different states corresponding to the 8 different lights on the pod, you would divide the difference of these two numbers by 8. When I did this, the result was 14.375. That means that the lighting of the next light meant an increase of 14.375 units from the analog to digital converter. So the first if statement in the program looks to see if the value `lightLevel` (our current level of light) is greater than the 14.375 of the range 25 to 140. That would be greater than 140-14.375 or 125.625. Since we are comparing an integer value, I rounded my readings to the nearest whole number. If the value of `lightLevel` is greater than 126, then `barData` will be equal to the number that will light all of the lights. That is B11111111. This is where using binary makes programming easy. We just put a 1 in the position of the light on the pod we want to turn on and a zero if we want it off. B11111111 is 255 in decimal numbers and is less easy to see that it is going to light all the lights.

The next level down for my setup would be 14.375 less than the last level. It is 111.25 (but I use the integer 111). Notice that the statement used here is “else if.” The reason for this is that when we reach the level that will light the lights we don’t want to continue evaluating the data. By using else if, this statement is only executed if the previous if or else if statement failed. If the previous if statement were true, this else if statement would be skipped. In the above code, if `lightLevel` were to have a value of 130, it would meet the first if statements condition (`lightLevel > 126`) and set `barData` to light all the lights. Since the next lines are else if statements, they will be skipped. If we just used the if statement, the next condition (`lightLevel > 111`) would also be true and `barData` would light all but the last light. That is not what we want, so we use the else if to stop looking after we have evaluated the correct `lightLevel` value.

Place this `loop()` code in your basic code setup. Make sure you set the values in the if statements to correspond to your own light level readings. Download and run the sketch. Now the Left Power Pod lights will show an increasing bar graph with light intensity.

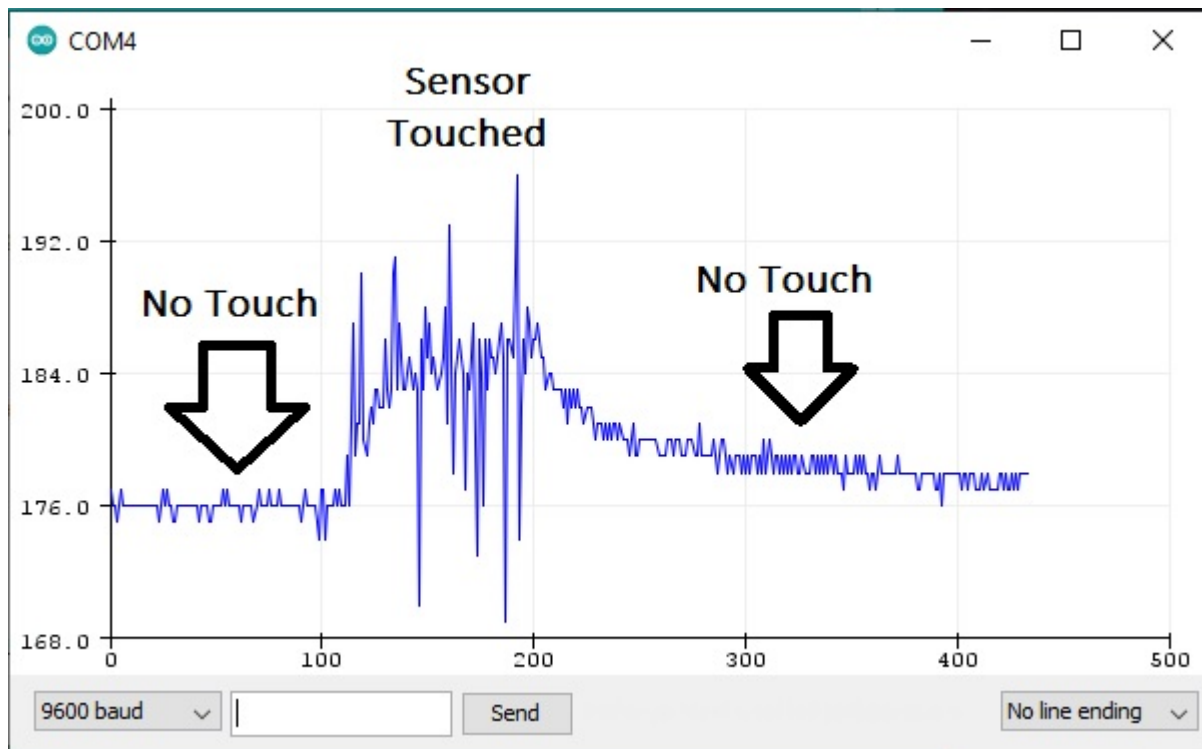
# The Temperature Sensor

The temperature sensor is a small microchip on the Starship that reports the temperature as an analog voltage. We will need to work out how this analog reading corresponds to the actual temperature. But before that, let's take a look at the raw signal coming from this sensor.

If we use the following code in our `loop()` routine:

```
float curTemp = analogRead(temperature_sensor);  
Serial.println(curTemp);  
dispNum(rpod_latch, curTemp);  
delay(10);
```

This code will load the current value of the temperature sensor into the variable `curTemp`. It will then send it to the Serial port and to the Right Power Pod lights. When you run this code and open the Serial Plotter in the Arduino software, you may see something like the following. Watch the sensor output. After a delay, touch the temperature sensor on the Starship. use a non-conductive material like a tissue or a piece of plastic wrap on your finger to prevent touching the leads of the sensor and changing the value. Your output may look something like the following :

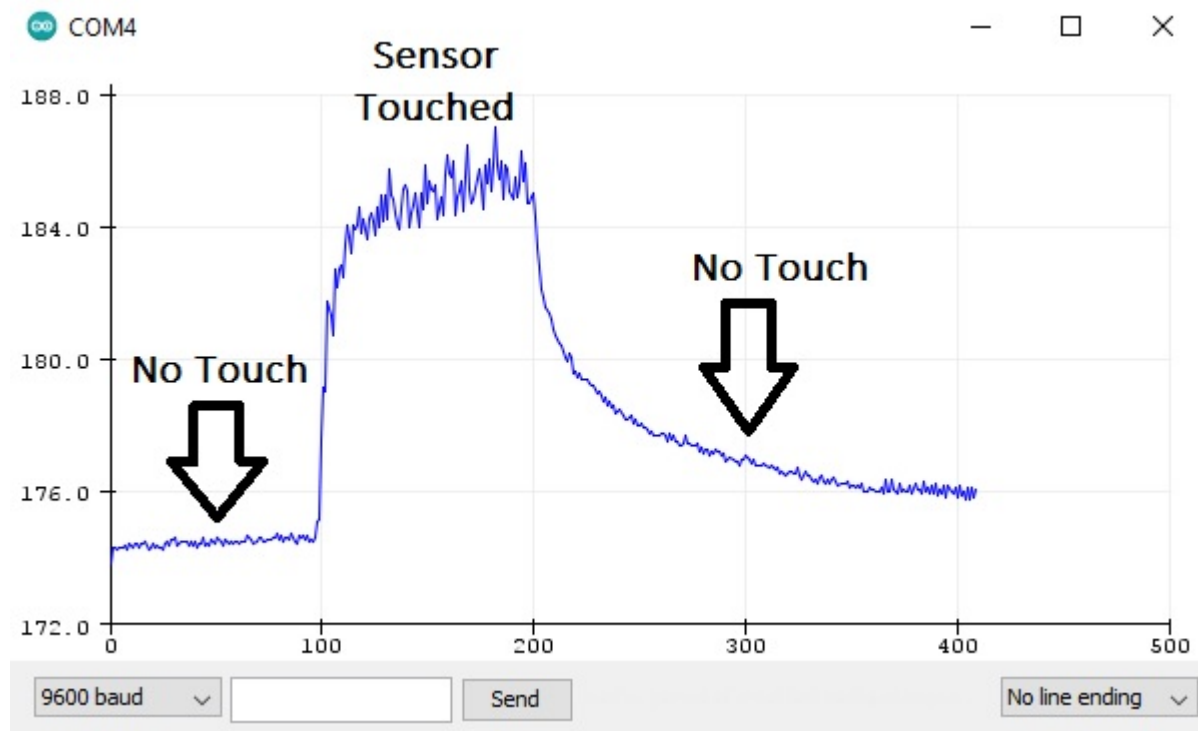


The graph shows some noise when the sensor is at rest. When you start to heat it up, you will see some large noise induced in the system. When you take your finger away, the noise will lower, but the signal is still pretty wild.

One way to smooth this signal is to take multiple readings and then average them to come up with a more smooth output. Use the following code in the loop() routine:

```
float curTemp = 0;
for (int i = 0; i < 50; i++) {
    curTemp = curTemp + analogRead(temperature_sensor);
    delay(10);
}
curTemp = curTemp / 50;
```

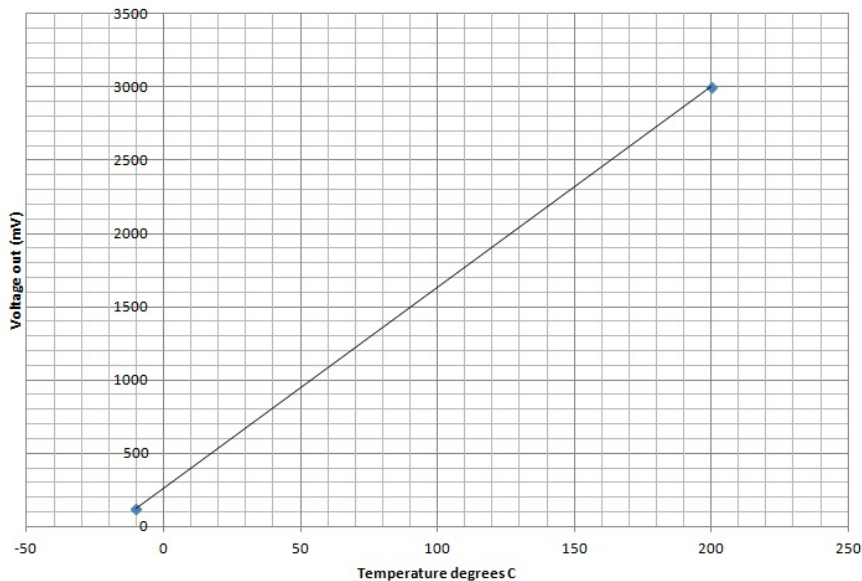
This code takes 50 readings, sums them, and then divides the result by 50 to get the average reading. When we do this, we can get a smoother output like the following:



You will notice that the signal is still a little noisy when the sensor is touched because of the interaction of your finger near the sensor, but the overall signal is much cleaner. Averaging is a typical way of smoothing an electronic signal to prevent wild spikes in the readings.

The graphs both showed the raw analog to digital converter. To be able to calculate the actual temperature from those, we can consult the data sheet for the sensor, the characteristics of our computers analog to digital converter, and some utilize some math.

From the sensor data sheet, the output voltage with respect to temperature is as follows:



The data sheet gives us two important points. The output should be 200 mV when the temperature is -10 degrees C. And the output should be 3000 mV when the temperature is 125 degrees C. Knowing two points allows us to come up with the equation of the line that connects them so we can determine the temperature from any voltage output reading from the sensor. The equation of a line is:

$$y = mx + b$$

Where m is the slope, and b is the y intercept. Since we know the value of x and y at two points on the line, let's create two equations that relate them.

$$3000 = m(125) + b$$

$$200 = m(-10) + b$$

Since we have two equations with two unknowns (m and b), we can actually solve these. Let's subtract the second equation from the first.

$$3000 - 200 = m(125 - (-10)) + b - b$$

Reducing the result gives:

$$2800 = 135m$$

This lets us solve for m easily.  $2800/135 = \mathbf{20.74}$ .

If we plug this value into either equation, we can calculate b.

$$200 = 20.74(-10) + b$$

$$\mathbf{b = 407.4}$$

So the equation of the line becomes:

$$y = 20.74x + 407.4$$

For our program, we will get the analog voltage and we want to convert it to a temperature reading. Since the x value represents temperature, we need to solve the equation for x:

$$x = \frac{y - 407.4}{20.74}$$

This will return the temperature for a voltage out of the sensor. The last thing we need to do is convert a voltage to the analog to digital conversion. The analog to digital converter measures voltage from 0 to 5 Volts and the converter gives a value from 0 to 1024. If we divide the voltage range by the digital value range we get  $5/1024 = 0.00488$  Volts per bit. This is also equal to 4.88 mV/bit.

Let's look at the following code:

```
float curTemp = 0;
for (int i = 0; i < 50; i++) {
    curTemp = curTemp + analogRead(temperature_sensor);
    delay(10);
}
float curT = curTemp / 50;
```

The above code samples the temperature sensor and gives the average of the 50 samples.

```
curT = curT * 4.88;
```

This line multiplies the reading by our conversion factor to make the reading into a voltage (in mV).

```
curT = (curT - 407.4) / 20.74;
```

This line is our equation that converts voltage (in mV) to degrees Celsius.

```
curT = curT * 9 / 5 + 32;
```

The above line converts the temperature to Fahrenheit.

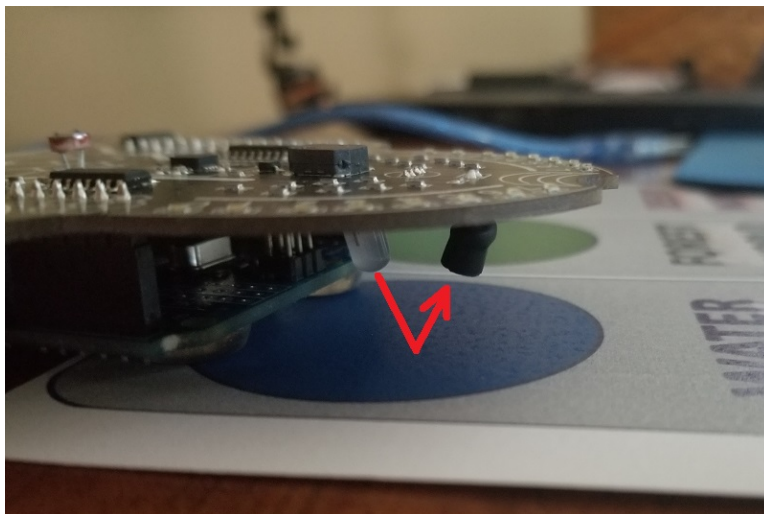
```
Serial.println(curT);  
dispNum(rpod_latch, curT);
```

And then we print the value to the Serial port and the right Power Pod lights. Download and run this program. Open the Serial Plotter to see the output. Experiment with the sensor. Try blowing on it and see what effect it has.

## The Color Sensor

When the Starship gets to planets, it wants to scan for water. For our purposes, if the planet is blue, we will consider it as having water. We need a color sensor.

The color sensor is located on the Disk underside. It is comprised of a tri-color LED and a light sensor. We will be placing colored paper under this sensor so it is a good time to make sure it is aligned. The following picture shows what the goal of the alignment is.



We want the LED to shine down on the paper and the light sensor to be aimed at the light being reflected. The light sensor is outfitted with a shield to prevent stray light from interfering. Take a moment to look and see that these two elements are aligned properly. If needed, place something under the Starship to lift it a little higher to allow more room for the LED and sensor to operate.

Consider adding rubber feet to the bottom of the Arduino. This will give a little more clearance to the color sensor and also prevent the Arduino from scratching your table.

The Starship has this hardware available. But first, we have to understand a little about color to see how we need to program it.

Color is actually just the frequency of visible light that we see. Our eyes pick up a small part of the electro-magnetic spectrum known as the visible light spectrum. What we call primary colors vary depending on whether we are talking about additive or subtractive color. In art class, we learn that Red, Yellow, and Blue are the primary colors. We can make all of the other colors by mixing these together in different ratios. However, you may have noticed that computer monitors are sometimes referred to as RGB monitors. That is because Red, Green, and Blue light can be used to make all other colors. It can get even more confusing when we go to buy printer ink and we see the “primary colors” here are Yellow, Magenta, and Cyan. Depending on how the light will interact with our base colors determines which set of “primary colors” we use. Our tri-color LED can emit Red, Green, or Blue light.

Light from the sun and many other sources are composed of the entire EM spectrum. That means they radiate all of the colors in the visible range. All of the colors mixed together gives the white appearance. When this light strikes an object, like a blue ball, some of the colors are absorbed and some are reflected. The reflected light is collected in our eyes and we see the colors. That means the blue ball absorbed every other color but blue. So is it a blue ball? Maybe it is an anti-blue ball? Either way, we see the reflected light. Let’s use this same concept to create a simple color detector with our Starship.

If we energize each of the tri-colored LEDs one at a time and look at the reflected light, we should be able to tell something about the color of the object we are over.

We used `#define` statements to map these LEDs to our Arduino pins.

```
#define led_red 9
#define led_blue 10
#define led_green 11
```

And the light sensor was defined as:

```
#define color_sensor A2
```

If we want to turn on an LED (say, the red one), we can just set that pin to HIGH.

```
digitalWrite(led_red, HIGH);
```



And we can set it to LOW to turn it off.

Let's write a simple sketch to see what effect different colors have on the output of the light sensor. As before, open your basic code framework that has all of the #define and setup routines we need. Save it under another name (like Starship\_color). Use the following code in your loop() routine:

```
void loop() {  
  
    digitalWrite(led_red, HIGH); // Turn on the red LED  
  
    for (int i = 0; i < 20; i++) {  
        Serial.println(analogRead(color_sensor));  
        delay(250);  
    }  
  
    digitalWrite(led_red, LOW); // Turn off the red LED  
  
    for (int i = 0; i < 20; i++) {  
        Serial.println(analogRead(color_sensor));  
        delay(250);  
    }  
}
```

The code simply turns on the red color sensing LED and reports the light value to the Serial port. Then it turns off the LED and reports the value again. We loop each reporting period 20 times and each has a 250 msec delay for a total of 5 seconds. Download the program to the Starship and open the Serial Monitor. Watch the difference in the readings with the LED on and off.

Write down the results in a table like the one below. We will look at the differences for each color LED and the different colors of reflective material. When you have completed all of the variations for the red LED, change the code so it will light the green LED and repeat. Do the same for the blue LED. When you are done, your table should look something like this:

Paper Color	ALL OFF	Red LED ON	Green LED ON	Blue LED ON
White	208	914	796	754
Black	23	496	286	252
Red	133	899	443	399
Green	46	633	519	372
Blue	43	590	482	571

Let's look at the results above which were taken on a Starship in our fleet (your Starship will have different numbers due to slight differences in manufacturing and ambient light conditions). We would expect that the white would reflect the greatest amount of light and the black would reflect the least. This is exactly what the data shows. For each color LED, the white returned the highest light level and the black produced the least. With the LEDs off, you would expect that the light level would be the same value for each test. However, because there is ambient light leaking into the area we are measuring, there are slight differences in the amount of reflected light because the light will be absorbed more or less depending on the color.

Let's show the data as a percentage each light return was compared to the highest and lowest it could be (the white and black respectively).

We can calculate that as:

$$percent = \frac{reading - black}{white - black} \times 100$$

Paper Color	ALL OFF	Red LED ON %	Green LED ON %	Blue LED ON %
Red	133	96.4	30.8	29.3
Green	46	32.8	45.7	23.9
Blue	43	22.4	38.4	63.5

This gives good correlation. For each color under the sensor, the colored LED had the highest percentage return. We could then use this sensor to detect the type of planet we are over and decide if we want to land. Let's write code to do that.

## Planet Scanner

We already have the code laid out for the scanner. We just need to use some variables to hold all of this data to be able to compare the results at the end.

First we need to get the calibration data. This is the color light returns for the Black and White samples. This will set the minimum and maximum light levels we would expect.

```
long baseRedBlack = 0; // calibration value Red LED on Black
long baseRedWhite = 0; // calibration value Red LED on White
long baseGreenBlack = 0;
```

```

long baseGreenWhite = 0;
long baseBlueBlack = 0;
long baseBlueWhite = 0;

```

First, we set aside variables to hold the calibration data for each LED for both the Black and White calibration samples. Since we are averaging 20 readings, we use a long integer as the variable to be able to store the large value of the result. We want the calibration data code to run in the setup() routine so it only executes once. The loop() will be where we put the code that tests for color.

```

Serial.println("Ready..place BLACK target and press button");
Serial.println();

while (digitalRead(button_input)) {}

// Get Red value
digitalWrite(led_red, HIGH); // Turn on the red LED

for (int i = 0; i < 20; i++) {
    delay(100);
    baseRedBlack = baseRedBlack + analogRead(color_sensor);
}
baseRedBlack = baseRedBlack / 20;

```

Here, we prompt the user to place the Black target under the sensor and press the button. We then average 20 samples of the color sensor and set it equal to the calibration variable (baseRedBlack in this run). We can then repeat this code for each color and then for the White calibration target.

```

long redVal = 0;
long greenVal = 0;
long blueVal = 0;

```

We then enter the loop() routine. We need three more variable to hold the individual color values. Wait for the user to press the button and then sample the light return, just like we did above. Repeat for each color.

When the color light values have been captured, calculate the percentages that the returned light is within the Black and White band.

$$percent = \frac{reading - black}{white - black} \times 100$$

```

// Calculate the percentages
redVal = 100 * (redVal - baseRedBlack) / (baseRedWhite -
baseRedBlack);
greenVal = 100 * (greenVal - baseGreenBlack) / (baseGreenWhite -
baseGreenBlack);
blueVal = 100 * (blueVal - baseBlueBlack) / (baseBlueWhite -
baseBlueBlack);

Serial.print("Red = ");
Serial.print(redVal);
Serial.println(" %");
Serial.print("Green = ");
Serial.print(greenVal);
Serial.println(" %");
Serial.print("Blue = ");
Serial.print(blueVal);
Serial.println(" %");

```

Calculate the percentages and print the results to the Serial port. Finally, look for the largest percentage and report whether the computer senses a Water, Forest, or Desert planet.

```

if (redVal > greenVal & redVal > blueVal)
Serial.println("Red...Desert World Found.");

else if (greenVal > redVal & greenVal > blueVal)
Serial.println("Green...Forest World Found.");

else Serial.println("Blue...Water World Found.");
Serial.println();

```

After this, the loop will repeat and it will prompt the user for another run.

Download and run the application and follow the prompts. After doing the calibration runs on the Black and White samples, place one of the colored planets under the sensor. Make sure the light from the LED lands on the planet color and the sensor is pointing at it. If the Starship is having problems identifying colors, try to gently adjust the LED and light sensor so the LED points its light where the sensor is viewing. From our initial data, you can see the Green LED had the smallest margin of the colors and is the most sensitive to error. Think of some ways that you can make this sensor more accurate.

Now that we have found a planet, let's land on it!

# Planet Descent and Landing

To code the dynamics of landing a spaceship, we will have to turn to some applied math and physics.

Physics defines displacement and the distance between two points. If you walk a mile to the east, your displacement would be 1 mile east.

Velocity is defined as the rate of change of displacement. In other words, how quickly am I moving from point A to point B. If we walk 1 mile east in 30 minutes, we would walk at a velocity of 2 miles per hour to the east.

Acceleration is defined as the rate of change of velocity. When you travel in a car, you may be going 55 miles per hour, but it doesn't feel like you are moving until you accelerate. That is, you speed up or slow down. When you hit the brakes, you feel yourself lift a little off your seat. We usually don't feel position, displacement, or velocity. But we feel acceleration. When something is accelerating, its velocity is changing.

The intention of this section is not to teach calculus, but to show the power of this field of mathematics.

Rate of change is one of the most powerful computational tricks of calculus. Think about it for a moment. When you are traveling in your car at 55 mph, you can look at your speedometer and know you are traveling at that speed. But if you use elementary physics to try to solve for your instantaneous velocity, you would find that the equation becomes impossible.

From physics:

$$speed = \frac{\text{change in distance}}{\text{change in time}}$$

If we go 1 mile in 30 minutes (0.5 hours) we can solve for the speed.

$$speed = \frac{1 \text{ mile}}{0.5 \text{ hour}} = 2 \text{ mph}$$

But what if we want instantaneous speed? At a single point in time, the change in time is 0 and the distance traveled is 0. So the equation becomes:

$$speed = \frac{0 \text{ miles}}{0 \text{ hours}} = ???$$

We know from algebra that division by 0 is undefined. And yet, when we travel in our car at 55 mph with the cruise control on, we know that at every time we look at the speedometer, it says 55 mph. We read the instantaneous speed, and know we are traveling at 55 mph but we cannot calculate it.

That is, unless we use calculus. Calculus lets us actually figure out instantaneous rates of change. This is amazingly important in science and engineering.

A Greek delta symbol is used to represent a change of something. This rate of change is called the *derivative*. So from the above definitions of displacement (d), velocity (v), acceleration (a), we can write them as calculus expressions.

$$\text{displacement} = d$$

$$\text{velocity} = v = \frac{\delta d}{\delta t}$$

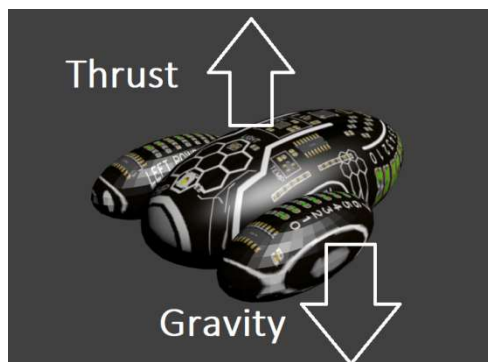
This equation states that velocity is the change of displacement over the change in time.

$$\text{acceleration} = a = \frac{\delta v}{\delta t}$$

This equation states that the acceleration is the change of velocity over the change in time.

Now with all maths, if there is a way to do something, there is another way to undo it. If you add something, you can subtract it. If you multiply something, you can divide it. And if you take the derivative of something, you can integrate to undo it. We can get velocity by taking the integral of acceleration with respect to time. And we can get displacement by integrating velocity over time. So let's start with acceleration and derive the ballistics equations we need to land our Starship.

Let's look at a free *body diagram* of our ship as it enters the planets gravitational field. A free body diagram is a picture that shows all of the forces on a body so we can develop physics equations.



We will neglect the friction of the atmosphere so the acceleration the Starship will feel is just gravity pulling down and thrust, when utilized, pushing up. We can then write the equation for acceleration of the Starship.

$$a_{total} = -g + a_{thrust}$$

In the equation,  $g$  is the gravitational acceleration ( $9.8 \text{ m/s}^2$ ) and  $a_{thrust}$  is the acceleration upward when the entry rockets are being fired (we will use  $30 \text{ m/s}^2$ ).

To program our Starship, we need to know the velocity and displacement. The velocity will be displayed on the disk LEDs while the displacement will be shown in the Serial Plotter of the Arduino software.

This is where calculus does its magic. By taking the integral of acceleration, we can derive the formula for velocity.

$$v = \int a \delta t = a_{total} t + v_o$$

Where  $v_o$  is the starting velocity (not all objects start at zero velocity). The term  $t$  is the change in time from initial velocity to current velocity. This formula will allow us to calculate the new velocity from the last velocity and the acceleration term.

We can now derive the displacement from integrating the velocity.

$$d = \int v \delta t = \frac{-a_{total} t^2}{2} + v_o t + d_o$$

These equations may look scary, but we know all of the information to get from acceleration, to velocity, to displacement. Let's look at what this looks like in code:

```
Accl = -9.8 + butPush * 30;  
Vel2 = Accl * deltaT + Vel1;  
yPos = Accl * (deltaT * deltaT) / 2 + Vel1 * deltaT + yPos;
```

The `Accl` variable will hold the current acceleration. Just like the equation we derived from the free body diagram, it is  $-9.8 \text{ m/s}^2$  plus the  $30 \text{ m/s}^2$  from the thrust. Note that we multiply the thrust by a variable called `butPush`. If the user pushes the button, we will assign a 1 to this variable. That will add the  $30 \text{ m/s}^2$  to the equation. If the button is not pushed (or the user runs out of fuel), we will assign 0 to `butPush` so you will not add the positive thrust to the overall acceleration.

We want the Starship to report current velocity and displacement (distance) above the planet as well as the remaining fuel. This will require some new display functions.

We will use the Power Pods to display the fuel remaining. This will require all the lights to be on initially, and turn off as the fuel is consumed.

```
void fuelLevel(byte curLevel) {
    byte lpod = 0;
    byte rpod = 0;
    byte vals[9] = {0, 1, 3, 7, 15, 31, 63, 127, 255};
    if (curLevel < 9) {
        lpod = 0;
        rpod = vals[curLevel];
    } else {
        rpod = B11111111;
        lpod = vals[curLevel - 8];
    }
    dispNum(lpod_latch, lpod);
    dispNum(rpod_latch, rpod);
}
```

This subroutine takes a byte value as input. That value can be 16 (full tank) to 0 (empty). The subroutine establishes an array of values. An array is a set of variables that can be accessed by an index.

```
byte vals[9] = {0, 1, 3, 7, 15, 31, 63, 127, 255};
```

This line establishes the array “vals” of type byte. The number in brackets (9) sets the size, or dimension, of the variable. It means that vals will have 9 items in it. We then assign the values with the bracketed numbers. To access a particular number, we will just use the index number. The array index starts at 0 and goes until the end of the array. For instance, vals[3] will return the number 7. This array holds the values that will light none to all of the lights in a bar graph type of display. Continuing in the code, we check to see if the number is less than 9. If that is true, we are out of fuel in the left pod and are drawing fuel from the right. Otherwise we are still working off the fuel in the left pod. We then end the subroutine by displaying the results on the Power Pods.

The second display routine we need is the velocity display on the disk. We will use the left disk lights to display negative velocity and the right for the positive velocity.

```
void diskVelocity(double curVel) {
    byte vals[9] = {0, 1, 3, 7, 15, 31, 63, 127, 255};
```



```

byte dispV = 0;
boolean neg = false;
if (curVel < 0) {
    neg = true;
    curVel = curVel * -1.0;
}
for (int i = 1; i < 9; i++) {
    if (curVel >= i * 10.0) dispV = vals[i];
}
if (neg) {
    dispNum(disk_left_latch, dispV);
    dispNum(disk_right_latch, 0);
} else {
    dispNum(disk_right_latch, dispV);
    dispNum(disk_left_latch, 0);
}
}

```

Since we want to display the velocity as a growing bar graph, we will use the same technique we used for the fuel display. We establish the array of needed values in the first line. We then check to see if the velocity is negative. If it is, we set a variable to track that it is negative and we turn it positive to be able to work with it more easily. We then determine how many lights to light based on each light being 10 m/s of velocity. Finally, we display the values on the disk lights.

We will need to display the altitude of the Starship. If we send the current altitude to the Serial port, we can use the plotter in the Arduino software to plot the position of the Starship. Since the plotter adjusts its x and y plot ranges based on the data it is sent, we will first send a zero value to set the lower bound of the plot. The next value sent will be the initial altitude. This will be 10 kilometers. This will set the max and min for the plotter and give us a stable display.

We will need to check for touchdown (or crash). The following code will accomplish this.

```

// Check for landing
if (yPos < 1.0) {
    if (Vel1 > -15.0) {
        // Safe landing
        safeLand();
    } else {
        // Crash
        crashLand();
    }
}

```

```

    while (1) {} // Stop here
}

```

First, we check to see if the Starship has reached the ground ( $yPos < 1$ ). Then we look at the velocity to see if we landed or crashed. A velocity less than 15 m/s is survivable. Any velocity greater than that is considered a crash. We left placeholders for what to do if we land successfully or crash. Let's deal with them.

```

void safeLand() {
    // Turn all lights off
    dispOff();
    delay(500);
    for (int i = 0; i < 17; i++) {
        fuelLevel(i);
        delay(100);
    }
    delay(500);
    diskSweep(50);
    tone(disk_speaker, 500, 200);
    diskSweep(50);
    tone(disk_speaker, 700, 200);
    dispNum(disk_right_latch, 7);
    dispNum(disk_left_latch, 7);
}

```

For the safe landing, we start by cycling through the fuel lights, we sweep the disk lights twice, play a quick success tone, and light the green disk lights.

```

void crashLand() {
    // Sad light display
    dispOff();
    delay(500);
    for (int i = 16; i >= 0; i--) {
        fuelLevel(i);
        delay(200);
    }
    delay(500);
    tone(disk_speaker, 400, 500);
    dispNum(disk_left_latch, 192);
    dispNum(disk_right_latch, 192);
}

```

When we crash, we turn off all lights, cycle the fuel up, play a sad tone, and light the red disk lights.

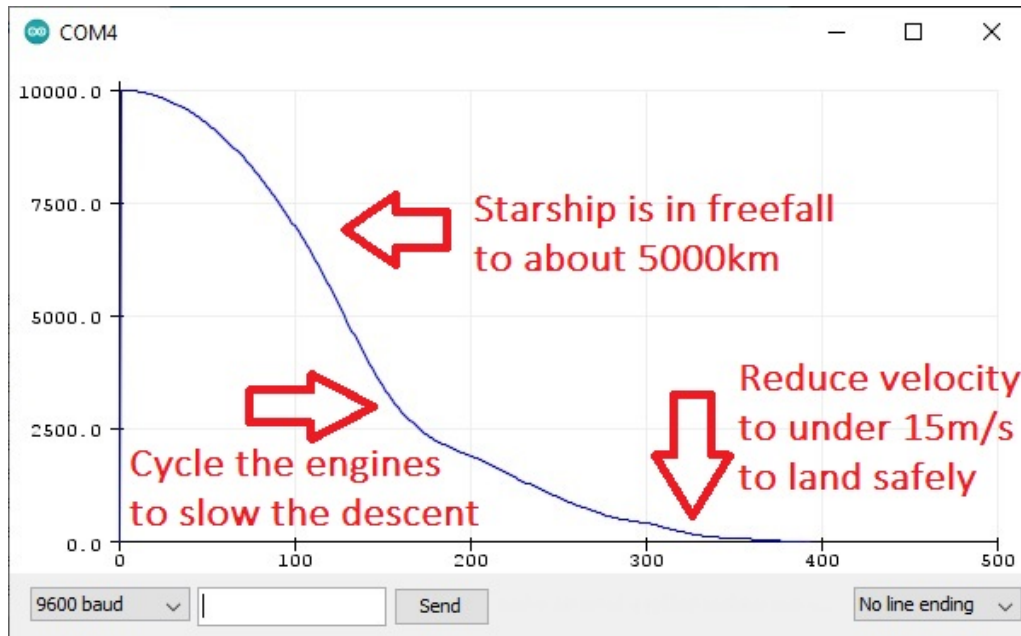
Load the program onto the Starship and open the Serial Plotter. You will see the following:

- 1) The plotter will show a graph of the height of the Starship above ground.
- 2) The left disk lights will start to light indicating that the Starship has turned off its gravity compensators and is now falling to the ground in accordance with the equations we calculated. These lights represent 10 m/sec of velocity each.
- 3) The Left and Right Power Pods have all lights lit indicating that the energy storage tanks are full.

At this point, the mission profile for landing looks like this:

- 1) Let the Starship fall until it gets to about 5000 km.
- 2) At about 5000 km, press the button to energize the landing thrusters.
- 3) Keep the thrusters on until you see the velocity lights come back into the red zone. They have been pegged high during the rapid descent to 5000 km.
- 4) Now, cycle the button to keep the descent velocity in the red or yellow zones as you continue to descend.
- 5) As you approach the landing, cycle the thrusters to continue to slowly reduce your velocity.
- 6) We need to land with  $<15$  m/sec negative velocity. Anymore, and the landing struts may collapse and the Starship may be damaged.
- 7) Continue to cycle the thrusters so you keep velocity below 10m/sec (one green light on). As soon as the green light turns on, press the button to reduce speed to have no lights lit. Do not over thrust or the Starship will start to gain altitude. Any positive velocity value will be shown on the right disk lights.
- 8) Continue to allow the Starship to slowly descend to touchdown. Watch your fuel level!
- 9) At touchdown, an animation will be displayed to signify you landed safely or you crashed.

After you complete a run, just close and restart the Serial Plotter to begin again. Below is a sample landing showing the key phases of the descent.



After a landing, look at the fuel you have left. Could you have slowed earlier and had a gentler descent? How far can you fall before full thrusters can't stop your ship from crashing? Can you determine the most efficient landing profile (the one that has the greatest fuel remaining)?

Play around and see what you find out. You can also change some of the parameters in the software. What happens if you double the value of gravity? How about halving the thrust? How can you give yourself unlimited fuel?

Now let's take a look at another sensor installed on our Starship. It is a force sensing device called an accelerometer.

# The Accelerometer

The Starship is a sophisticated ship with sensors to navigate and explore the solar system and beyond. One of the key sensors is an accelerometer. An accelerometer measures instantaneous acceleration of an object. As shown in the previous section with our use of calculus, if we know acceleration, we can determine velocity and distance.

Let's access the accelerometer and use it to display the tilt of the Starship. First, let's look how this device measures acceleration.

In a way, you act as an accelerometer when you travel in a car. Your body moves independent of the car so when the car speeds up (accelerates), you move back against the seat. When the car brakes (decelerates), you lift off the seat slightly. When the car turns, your body tends to move in the opposite direction. But when the car is traveling at a constant velocity (acceleration is zero), you don't feel as if you are moving. This is all because of Newton's Laws of Motion. Newton's First Law states that a body in motion stays in motion and a body at rest stays at rest unless acted upon by a force. His second law states that force is equal to mass times acceleration. With a constant mass, that means that the force you feel in the car that pushes you around is due to the acceleration of the car. Acceleration is a vector quantity which means it has a size and direction. Your speed in the car may stay the same, but when you turn, you are changing the direction of that speed and that takes acceleration too. Speeding up, slowing down, and turning are all accomplished with acceleration.

There are many types of accelerometers. Many accelerometers used in the electronics industry use the same principle as the person being thrown around in the car. They use a stable structure with a moving piece and measure the amount of deflection the moving piece experiences. This can be related to the acceleration.

The accelerometer in use on the Starship is a 3-axis, analog output accelerometer. The device measures the deflection of an internal structure, converts it to a voltage and makes it available on the x, y, and z output pins. We can then use the analog to digital converter on the Starship to access these values.

Just like the light sensors, we access the output value of the accelerometer with the `analogRead()` subroutine.

```
analogRead(accl_X) ;  
analogRead(accl_Y) ;  
analogRead(accl_Z) ;
```

Acceleration in the x, y, and z directions may be positive or negative. The Arduino ports do not read negative voltages, so the accelerometer at rest produces a positive value that represents zero acceleration. The number gets larger for positive accelerations and smaller for negative accelerations.

Let's make a tilt display using the accelerometer and the Starship's lights. The Power Pod lights will show the tilt forward and back and the disk lights will show the tilt from right to left.

We could sample the resting accelerations and use that as the zero points for acceleration. The problem is that we don't know what the maximum and minimum values are. Because of this, we wouldn't know how to scale the lights so they would use the whole range of acceleration. One way to do this is to calibrate the accelerometer first. We can write code to sample the accelerometer readings and look for the maximum and minimum while the Starship is being moved through 360 degrees in the x and y axes.

First we will declare variables to hold the max and min information. This will happen before the setup() and loop() routines so they will be available to all routines.

```
int maxX = 0;
int maxY = 0;
int maxZ = 0;
int minX = 1024;
int minY = 1024;
int minZ = 1024;
```

By declaring the variables this way, when the program starts and we compare them to the actual readings, we are assured to be larger than the minimum and smaller than the maximum so the first readings will set the new min and max values.

We then prompt the user to rotate the Starship around.

```
Serial.println("Move the Starship through 360 degrees");
Serial.println("Press the button when done");
```

```
int tempAcc;
while (digitalRead(button_input)) {
    tempAcc = analogRead(accl_X);
    if (tempAcc > maxX) maxX = tempAcc;
    if (tempAcc < minX) minX = tempAcc;
    tempAcc = analogRead(accl_Y);
    if (tempAcc > maxY) maxY = tempAcc;
    if (tempAcc < minY) minY = tempAcc;
```

```

    tempAcc = analogRead(accl_Z);
    if (tempAcc > maxZ) maxZ = tempAcc;
    if (tempAcc < minZ) minZ = tempAcc;
}
Serial.println("Calibration Complete.");

```

While the Starship is being moved, the while loop continually samples the acceleration values and if a value is less than the minimum or greater than the maximum it sets the global variables to the new value. When we exit this while loop (by pressing the button) we have the max and min values stored. Now we know the extent of the readings we can scale the lights to cover the expected range.

Let's pass off the task of displaying values on the disk and pods to subroutines. This will make our loop() code really easy. Let's start with the X-axis values and the Power Pod lights.

```

void acclLevelX(int curLevel) {
    byte vals[9] = {0, 1, 2, 4, 8, 16, 32, 64, 128};
    if (curLevel > maxX) curLevel = maxX;
    if (curLevel < minX) curLevel = minX;
    int diffX = (curLevel - minX) / ((maxX - minX) / 8);
    dispNum(lpod_latch, vals[diffX]);
    dispNum(rpod_latch, vals[diffX]);
}

```

As before, we choose a name that makes sense (acclLevelX). We are passing the value of the accelerometer output to the subroutine so we will pass the value into an integer named curLevel. The first thing we want to do is constrain the value to the max and min values. If we shake the Starship, we will generate much larger values than by just tilting it. This could cause us to try to access the vals[] array with too large or too small (a negative) number so we check to see if the value is greater than the max or less than the min. If it is, we set the curLevel to that max or min. We only want to light one light at a time so we establish an array to be able to convert the numbers 1 thru 8 to binary values that light just that number LED. We then need to compare the current value to the maximum and minimum to determine which of the 8 LEDs to light. We do this by finding out what one eighth of the total range is ( (maxX-minX)/8 ). We can then divide this into the current value ( curLevel - minX ) to see how many of these divisions the value represents. We can then use this number to light the associated pod lights.

We use the same process for the Y value.

```

void acclLevelY(int curLevel) {

```

```

byte vals[9] = {0, 1, 2, 4, 8, 16, 32, 64, 128};
if (curLevel > maxX) curLevel = maxX;
if (curLevel < minX) curLevel = minX;
int diffY = (curLevel - minX) / ((maxX - minX) / 16);
dispNum(disk_left_latch, 0);
dispNum(disk_right_latch, 0);
if (diffY < 9) dispNum(disk_right_latch, vals[diffY]);
else dispNum(disk_left_latch, vals[17 - diffY]);
}

```

The only real differences are that we are lighting the disk lights and we have to check if we are in the left or right side.

With subroutines doing the heavy lifting, our loop routine just has to collect the X and Y values and pass them to the display subroutines.

Download and run the sketch. Open the Serial Monitor. You will be prompted to tilt the Starship. Gently move the Starship around so the accelerometer can measure the max and min values. Move it at least 180 degrees in the X and Y directions (side to side and tilt up and down). When you are complete, set the Starship down and press the button. Now as you rotate the Starship, the lights will show the relative tilt on the Starship.

You might notice that when the Starship is laying flat, the lights show a little tilt. This might be due to a slight difference in the tilt you did one way or the other. Or you may have jerked it in one direction which made the readings a little lop sided. How could you fix this? One way might be to sample the values when you put the Starship back down and press the button. These would give a center value. You could then use the difference between the center value and the max as your min and max for the positive tilt and the difference between the center and min for the negative tilt. Experiment with the accelerometer to get familiar with this useful sensor.

## Collision Detection

Another use of the accelerometer is to detect unexpected events. As we cruise through space, we should not feel any motion of the ship as long as it is not accelerating. If the ship strikes space debris or is shot with a laser, we want to generate some automatic actions like sounding an alarm and putting up shields. We can monitor the accelerometer output and trigger the alarm if it gets outside of a desired value. Let's call that value:

```
int sens = 5;
```



This variable will be used to compare the resting value of the accelerometer to the current value. If the current value gets outside of the sens value, we know that acceleration has occurred. We can then trigger the red disk lights and create a klaxon-like sound.

```
void alarmAnim() {
  for (int t = 0; t < 3; t++) {
    dispNum(disk_left_latch, 192);
    dispNum(disk_right_latch, 192);
    for (int i = 8; i >= 0; i--) {
      tone(disk_speaker, 600 - i * 20, 25);
      delay(25);
    }
    dispNum(disk_left_latch, 0);
    dispNum(disk_right_latch, 0);
    delay(200);
  }
}
```

This routine will light the red disk lights. It will then create a short, rising sound in the speaker. The red lights will be extinguished and the process will loop three times. This just leaves our loop() routine to check the values of the accelerometer and if it is outside of our tolerance (sens) then sound the alarm. Since we are looking for a change from the resting state, it is good to give a countdown to activation before enabling the alarm. The first part of the loop() routine will wait for a button press and then count down from 6 to 0 and then sample the resting values and enable monitoring for shock on the Starship. This gives the crew plenty of time to operate the switch (which could cause a slight acceleration in the sensors and thus, influence our resting readings) and let things settle out before arming.

```
void loop() {
  while (digitalRead(button_input)) {}
  for (int i = 0; i < 7; i++) {
    dispNum(lpod_latch, B00111111 >> i);
    delay(800);
  }
  int curX = analogRead(accl_X);
  int curY = analogRead(accl_Y);
  int curZ = analogRead(accl_Z);

  boolean alarm = false;

  while (!alarm) {
```

```

    if (analogRead(accl_X) > curX + sens) alarm = true;
    if (analogRead(accl_Y) > curY + sens) alarm = true;
    if (analogRead(accl_Z) > curZ + sens) alarm = true;
    if (analogRead(accl_X) < curX - sens) alarm = true;
    if (analogRead(accl_Y) < curY - sens) alarm = true;
    if (analogRead(accl_Z) < curZ - sens) alarm = true;
  }
  alarmAnim();
}

```

Download and run the code. When the ship is stable, press the button. You will see the timer count down on the Left Power Pod lights. When you reach zero, the alarm is active. Try bumping the table or lifting the Starship. You should not be able to do much before the alarm sounds. If the alarm is too sensitive or not sensitive enough, change the sens variable to change this sensitivity. That is one of the reasons for using a variable to control sensitivity. You only need to change the value in code once, and it will be updated all places in the code.

## What's next?

I don't know. It is up to you where you go from here. You have the fundamentals to build code, explore colors, sense movement and light, measure temperature and communicate with lights and sound. Experiment with modifying the programs we have created. Make the color program identify more colors. Make the lander program harder by increasing the gravity. Let your imagination take your Starship to great places.

Best of luck on your future missions!

# Appendix A

Software listings. The files can be downloaded at:

<https://github.com/Subsystems-us/Starship>

## Starship\_display

This is the basic code including #defines, setup(), and common display routines.

```
// This program includes all of the needed display functions,
// #defines, and setup code.
// You can use this as the starting point for many programs.

// Map the Arduino pins to the Starship functions
#define shift_data 3 // Data for the LED shift registers
#define shift_clk 4 // Clock for the LED shift registers
#define lpod_latch 5 // Register latch for the left Power Pod
#define rpod_latch 6 // Register latch for the right Power Pod
#define disk_left_latch 7 // Latch for the left disk LEDs
#define disk_right_latch 8 // Latch for the right disk LEDs
#define led_red 9 // Red LED for the color sensor
#define led_blue 10 // Blue LED for the color sensor
#define led_green 11 // Green LED for the color sensor
#define disk_speaker 12 // Speaker connection
#define button_input 13 // User button
#define temperature_sensor A0 // Temperature sensor input
#define light_sensor A1 // Light sensor input
#define color_sensor A2 // Color sensor input
#define accl_X A3 // Accelerometer analog X-axis signal
#define accl_Y A4 // Accelerometer analog Y-axis signal
#define accl_Z A5 // Accelerometer analog Z-axis signal

void setup() {
    // Start the Serial connection at 9600 baud
    Serial.begin(9600);

    // Set the pin Modes
    pinMode(led_red, OUTPUT);
```

```

pinMode(led_blue, OUTPUT);
pinMode(led_green, OUTPUT);
pinMode(shift_data, OUTPUT);
pinMode(shift_clk, OUTPUT);
pinMode(lpod_latch , OUTPUT);
pinMode(rpod_latch , OUTPUT);
pinMode(disk_left_latch, OUTPUT);
pinMode(disk_right_latch, OUTPUT);
pinMode(disk_speaker, OUTPUT);
pinMode(button_input, INPUT);

// Turn off color sensor LEDs
digitalWrite(led_red, LOW); // Turn off the red LED
digitalWrite(led_blue, LOW); // Turn off the blue LED
digitalWrite(led_green, LOW); // Turn off the green LED

dispOff(); // Turn off all register LEDs
}

void loop() {

void dispNum(char ledSet, char ledValue) {
    // set the desired latch low
    digitalWrite(ledSet, LOW);
    // shift out the desired number
    shiftOut(shift_data, shift_clk, MSBFIRST, ledValue);
    // set the desired latch high
    digitalWrite(ledSet, HIGH);
}

void dispPoint(char ledSet, char ledValue) {
    byte vals[9] = {0, 1, 2, 4, 8, 16, 32, 64, 128};
    dispNum(ledSet, vals[ledValue]);
}

void dispBar(char ledSet, char ledValue) {
    byte vals[9] = {0, 1, 3, 7, 15, 31, 63, 127, 255};
    dispNum(ledSet, vals[ledValue]);
}

```

```

void dispOff() {
    // Turn all lights off
    dispNum(rpod_latch, 0);
    dispNum(lpod_latch, 0);
    dispNum(disk_left_latch, 0);
    dispNum(disk_right_latch, 0);
}

// This will light the lights in sequence. ledSet is the set of
// LEDs to use,
// swpDir is true for forward, false for reverse. swpSpeed is
// the msec time
// between each LED.
void dispSweep(char ledSet, boolean swpDir, long swpSpeed) {
    int num = B10000000;
    if (swpDir) num = B00000001;

    for (int i = 0; i < 8; i++) {
        dispNum(ledSet, num);
        if (swpDir) num = num * 2;
        else num = num / 2;
        delay(swpSpeed);
    }
    dispNum(ledSet, 0);
}

// This will sweep the disk lights from forward to back.
void diskSweep(long swpSpeed) {
    int num = B10000000;

    for (int i = 0; i < 9; i++) {
        dispNum(disk_left_latch, num);
        dispNum(disk_right_latch, num);
        num = num / 2;
        delay(swpSpeed);
    }
}

```

## Starship\_BridgeLights

This code randomly lights the Starship lights.

```
// This program simulates the crazy lights on the random panels
// in old sci-fi shows

// Map the Arduino pins to the Starship functions
#define shift_data 3 // Data for the LED shift registers
#define shift_clk 4 // Clock for the LED shift registers
#define lpod_latch 5 // Register latch for the left Power Pod
#define rpod_latch 6 // Register latch for the right Power Pod
#define disk_left_latch 7 // Latch for the left disk LEDs
#define disk_right_latch 8 // Latch for the right disk LEDs
#define led_red 9 // Red LED for the color sensor
#define led_blue 10 // Blue LED for the color sensor
#define led_green 11 // Green LED for the color sensor
#define disk_speaker 12 // Speaker connection
#define button_input 13 // User button
#define temperature_sensor A0 // Temperature sensor input
#define light_sensor A1 // Light sensor input
#define color_sensor A2 // Color sensor input
#define accl_X A3 // Accelerometer analog X-axis signal
#define accl_Y A4 // Accelerometer analog Y-axis signal
#define accl_Z A5 // Accelerometer analog Z-axis signal

void setup() {
    // Start the Serial connection at 9600 baud
    Serial.begin(9600);

    // Set the pin Modes
    pinMode(led_red, OUTPUT);
    pinMode(led_blue, OUTPUT);
    pinMode(led_green, OUTPUT);
    pinMode(shift_data, OUTPUT);
    pinMode(shift_clk, OUTPUT);
    pinMode(lpod_latch, OUTPUT);
    pinMode(rpod_latch, OUTPUT);
    pinMode(disk_left_latch, OUTPUT);
    pinMode(disk_right_latch, OUTPUT);
    pinMode(disk_speaker, OUTPUT);
}
```

```

pinMode(button_input, INPUT);

// Turn off color sensor LEDs
digitalWrite(led_red, LOW); // Turn off the red LED
digitalWrite(led_blue, LOW); // Turn off the blue LED
digitalWrite(led_green, LOW); // Turn off the green LED

dispOff(); // Turn off all register LEDs
}

void loop() {
    dispNum(lpod_latch, random(256));
    delay(250);
    dispNum(rpod_latch, random(256));
    delay(100);
    dispNum(disk_right_latch, random(256));
    delay(350);
    dispNum(disk_left_latch, random(256));
    delay(200);
}

void dispNum(char ledSet, char ledValue) {
    // set the desired latch low
    digitalWrite(ledSet, LOW);
    // shift out the desired number
    shiftOut(shift_data, shift_clk, MSBFIRST, ledValue);
    // set the desired latch high
    digitalWrite(ledSet, HIGH);
}

void dispOff() {
    // Turn all lights off
    dispNum(rpod_latch, 0);
    dispNum(lpod_latch, 0);
    dispNum(disk_left_latch, 0);
    dispNum(disk_right_latch, 0);
}

```

## Starship\_planet\_run

This simulates a fly through the solar system. Hang on!

```
// This program simulates flying through the solar system from
the sun to Neptune.
// The Starship shields skip on the planets atmosphere when it
passes and displays
// the name of the planet in the monitor.

// Map the Arduino pins to the Starship functions
#define shift_data 3 // Data for the LED shift registers
#define shift_clk 4 // Clock for the LED shift registers
#define lpod_latch 5 // Register latch for the left Power Pod
#define rpod_latch 6 // Register latch for the right Power Pod
#define disk_left_latch 7 // Latch for the left disk LEDs
#define disk_right_latch 8 // Latch for the right disk LEDs
#define led_red 9 // Red LED for the color sensor
#define led_blue 10 // Blue LED for the color sensor
#define led_green 11 // Green LED for the color sensor
#define disk_speaker 12 // Speaker connection
#define button_input 13 // User button
#define temperature_sensor A0 // Temperature sensor input
#define light_sensor A1 // Light sensor input
#define color_sensor A2 // Color sensor input
#define accl_X A3 // Accelerometer analog X-axis signal
#define accl_Y A4 // Accelerometer analog Y-axis signal
#define accl_Z A5 // Accelerometer analog Z-axis signal

void setup() {
  // Start the Serial connection at 9600 baud
  Serial.begin(9600);

  // Set the pin Modes
  pinMode(led_red, OUTPUT);
  pinMode(led_blue, OUTPUT);
  pinMode(led_green, OUTPUT);
  pinMode(shift_data, OUTPUT);
  pinMode(shift_clk, OUTPUT);
  pinMode(lpod_latch, OUTPUT);
  pinMode(rpod_latch, OUTPUT);
```



```

pinMode(disk_left_latch, OUTPUT);
pinMode(disk_right_latch, OUTPUT);
pinMode(disk_speaker, OUTPUT);
pinMode(button_input, INPUT);

// Turn off color sensor LEDs
digitalWrite(led_red, LOW); // Turn off the red LED
digitalWrite(led_blue, LOW); // Turn off the blue LED
digitalWrite(led_green, LOW); // Turn off the green LED

dispOff(); // Turn off all register LEDs

Serial.println("Ready for launch ...");

while (digitalRead(button_input)) {}
Serial.println("Start...");

delay(796);
diskSweep(25);
dispNum(rpod_latch, 1);
dispNum(lpod_latch, 1);
Serial.println("Mercury");
tone(disk_speaker, 500, 250);

delay(454); //We subtract the time to diskSweep (654-200)ms
diskSweep(25); // 25ms x 8 leds is 200ms
dispNum(rpod_latch, 2);
dispNum(lpod_latch, 2);
Serial.println("Venus");
tone(disk_speaker, 550, 250);

delay(360);
diskSweep(25);
dispNum(rpod_latch, 4);
dispNum(lpod_latch, 4);
Serial.println("Earth");
tone(disk_speaker, 600, 250);

delay(850);
diskSweep(25);
dispNum(rpod_latch, 8);

```

```

dispNum(lpod_latch, 8);
Serial.println("Mars");
tone(disk_speaker, 650, 250);

delay(6940);
diskSweep(25);
dispNum(rpod_latch, 16);
dispNum(lpod_latch, 16);
Serial.println("Jupiter");
tone(disk_speaker, 700, 250);

delay(9600);
diskSweep(25);
dispNum(rpod_latch, 32);
dispNum(lpod_latch, 32);
Serial.println("Saturn");
tone(disk_speaker, 750, 250);

delay(19500);
diskSweep(25);
dispNum(rpod_latch, 64);
dispNum(lpod_latch, 64);
Serial.println("Uranus");
tone(disk_speaker, 800, 250);

delay(20100);
diskSweep(25);
dispNum(rpod_latch, 128);
dispNum(lpod_latch, 128);
Serial.println("Neptune");
tone(disk_speaker, 850, 250);
}

void loop() {

void dispNum(char ledSet, char ledValue) {
    digitalWrite(ledSet, LOW);
    shiftOut(shift_data, shift_clk, MSBFIRST, ledValue);
    digitalWrite(ledSet, HIGH);
}

```

```

void dispOff() {
    // Turn all lights off
    dispNum(rpod_latch, 0);
    dispNum(lpod_latch, 0);
    dispNum(disk_left_latch, 0);
    dispNum(disk_right_latch, 0);
}

void diskSweep(long swpSpeed) {
    int num = B10000000;

    for (int i = 0; i < 9; i++) {
        dispNum(disk_left_latch, num);
        dispNum(disk_right_latch, num);
        num = num / 2;
        delay(swpSpeed);
    }
}

```

## Starship\_better\_binary

This sketch displays a binary number on the Left Power Pod. The user then enters a guess at the decimal value of the number. The sketch displays the guess on the Right Power Pod, checks if it is right and shows a “correct” or “wrong” animation.

```
// This program displays a binary number on the Left Power Pod
// The user then enters a decimal number into the serial monitor
// The computer then displays whether the user converted the
// number correctly or not.
```

```
// Map the Arduino pins to the Starship functions
#define shift_data 3 // Data for the LED shift registers
#define shift_clk 4 // Clock for the LED shift registers
#define lpod_latch 5 // Register latch for the left Power Pod
#define rpod_latch 6 // Register latch for the right Power Pod
#define disk_left_latch 7 // Latch for the left disk LEDs
#define disk_right_latch 8 // Latch for the right disk LEDs
#define led_red 9 // Red LED for the color sensor
#define led_blue 10 // Blue LED for the color sensor
#define led_green 11 // Green LED for the color sensor
#define disk_speaker 12 // Speaker connection
#define button_input 13 // User button
#define temperature_sensor A0 // Temperature sensor input
#define light_sensor A1 // Light sensor input
#define color_sensor A2 // Color sensor input
#define accl_X A3 // Accelerometer analog X-axis signal
#define accl_Y A4 // Accelerometer analog Y-axis signal
#define accl_Z A5 // Accelerometer analog Z-axis signal
```

```
void setup() {
```

```
    // Start the Serial connection at 9600 baud
    Serial.begin(9600);
```

```
    randomSeed(analogRead(light_sensor)); // Randomize our number
    generator
```

```
    // Set the pin Modes
    pinMode(led_red, OUTPUT);
    pinMode(led_blue, OUTPUT);
```

```

pinMode(led_green, OUTPUT);
pinMode(shift_data, OUTPUT);
pinMode(shift_clk, OUTPUT);
pinMode(lpod_latch , OUTPUT);
pinMode(rpod_latch , OUTPUT);
pinMode(disk_left_latch, OUTPUT);
pinMode(disk_right_latch, OUTPUT);
pinMode(disk_speaker, OUTPUT);
pinMode(button_input, INPUT);

// Turn off color sensor LEDs
digitalWrite(led_red, LOW); // Turn off the red LED
digitalWrite(led_blue, LOW); // Turn off the blue LED
digitalWrite(led_green, LOW); // Turn off the green LED

dispOff(); // Turn off all register LEDs

Serial.println("Ready to play ...");
}

void loop() {
  // Turn all lights off
  dispOff();

  int binaryNum = random(256);

  // Display it on the Left Power Pod
  dispNum(lpod_latch, binaryNum);

  // Prompt the user for the guess
  Serial.println("Enter your guess.");
  byte myGuess = getSerialData();

  // display the guess on the Right Pod
  dispNum(rpod_latch, myGuess);
  delay(2000);

  // Did we get it right?
  if (myGuess == binaryNum) {
    // We were right!
    diskSweep(50);
  }
}

```

```

        dispNum(disk_right_latch, B00000111); // Display Green
Lights
        dispNum(disk_left_latch, B00000111);
        Serial.println("Correct!!");
    } else {
        dispNum(disk_right_latch, B11000000); // Display Red Lights
        dispNum(disk_left_latch, B11000000);
        tone(disk_speaker, 500, 1000); // sound sad tone
        Serial.print("Sorry... the correct number was ");
        Serial.println(binaryNum);
    }
    delay(3000); // Allow the user to see the results for 3
seconds
}

byte getSerialData() {
    char receivedChars[4]; // an array to store the received
data
    char charNum = 0; //Track which digit has been received
    boolean notDone = true; // Allow us to exit when the data is
done
    char rc; // This will hold the character received

    while (Serial.available() == 0) {} // Wait until Serial data
appears

    while (Serial.available() > 0 && notDone) { // while there is
Serial data ...
        rc = Serial.read();

        delay(100); // This makes sure we don't miss data

        if (rc != '\n') {
            receivedChars[charNum] = rc;
            charNum++;
        }
        else {
            receivedChars[charNum] = '\0'; // terminate the string
            notDone = false;
        }
    }
}

```

```

    return atoi(receivedChars);
}

void dispNum(char ledSet, char ledValue) {
    // set the desired latch low
    digitalWrite(ledSet, LOW);
    // shift out the desired number
    shiftOut(shift_data, shift_clk, MSBFIRST, ledValue);
    // set the desired latch high
    digitalWrite(ledSet, HIGH);
}

void dispOff() {
    // Turn all lights off
    dispNum(rpod_latch, 0);
    dispNum(lpod_latch, 0);
    dispNum(disk_left_latch, 0);
    dispNum(disk_right_latch, 0);
}

void diskSweep(long swpSpeed) {
    int num = B10000000;

    for (int i = 0; i < 9; i++) {
        dispNum(disk_left_latch, num);
        dispNum(disk_right_latch, num);
        num = num / 2;
        delay(swpSpeed);
    }
}

```

## Starship\_light\_sensor

This sketch uses the output of the light sensor to light the Left Power Pod LEDs.

```
// This sketch uses the output of the light sensor to light
// the Left Power Pod LEDs

// Map the Arduino pins to the Starship functions
#define shift_data 3 // Data for the LED shift registers
#define shift_clk 4 // Clock for the LED shift registers
#define lpod_latch 5 // Register latch for the left Power Pod
#define rpod_latch 6 // Register latch for the right Power Pod
#define disk_left_latch 7 // Latch for the left disk LEDs
#define disk_right_latch 8 // Latch for the right disk LEDs
#define led_red 9 // Red LED for the color sensor
#define led_blue 10 // Blue LED for the color sensor
#define led_green 11 // Green LED for the color sensor
#define disk_speaker 12 // Speaker connection
#define button_input 13 // User button
#define temperature_sensor A0 // Temperature sensor input
#define light_sensor A1 // Light sensor input
#define color_sensor A2 // Color sensor input
#define accl_X A3 // Accelerometer analog X-axis signal
#define accl_Y A4 // Accelerometer analog Y-axis signal
#define accl_Z A5 // Accelerometer analog Z-axis signal

void setup() {

    // Start the Serial connection at 9600 baud
    Serial.begin(9600);

    randomSeed(analogRead(light_sensor)); // Randomize our number
    generator

    // Set the pin Modes
    pinMode(led_red, OUTPUT);
    pinMode(led_blue, OUTPUT);
    pinMode(led_green, OUTPUT);
    pinMode(shift_data, OUTPUT);
    pinMode(shift_clk, OUTPUT);
    pinMode(lpod_latch , OUTPUT);
```



```

pinMode(rpod_latch , OUTPUT);
pinMode(disk_left_latch, OUTPUT);
pinMode(disk_right_latch, OUTPUT);
pinMode(disk_speaker, OUTPUT);
pinMode(button_input, INPUT);

// Turn off color sensor LEDs
digitalWrite(led_red, LOW); // Turn off the red LED
digitalWrite(led_blue, LOW); // Turn off the blue LED
digitalWrite(led_green, LOW); // Turn off the green LED

dispOff(); // Turn off all register LEDs

Serial.println("Ready to play ...");
}

void loop() {
  byte barData = B00000000;
  int lightLevel = analogRead(light_sensor) / 4;

  if (lightLevel > 126) barData = B11111111;
  else if (lightLevel > 111) barData = B01111111;
  else if (lightLevel > 97) barData = B00111111;
  else if (lightLevel > 83) barData = B00011111;
  else if (lightLevel > 68) barData = B00001111;
  else if (lightLevel > 54) barData = B00000111;
  else if (lightLevel > 39) barData = B00000011;
  else if (lightLevel > 25) barData = B00000001;
  else barData = B00000000;

  dispNum(lpod_latch, barData);
  delay(100);
}

void dispNum(char ledSet, char ledValue) {
  // set the desired latch low
  digitalWrite(ledSet, LOW);
  // shift out the desired number
  shiftOut(shift_data, shift_clk, MSBFIRST, ledValue);
  // set the desired latch high
  digitalWrite(ledSet, HIGH);
}

```

```
}

void dispOff() {
    // Turn all lights off
    dispNum(rpod_latch, 0);
    dispNum(lpod_latch, 0);
    dispNum(disk_left_latch, 0);
    dispNum(disk_right_latch, 0);
}
```

## Starship\_color\_test

This sketch samples a color under the sensor and reports the probability of the color.

```
// This program samples the color of an object under
// the Starship and displays the probability of
// the detected color.

// Map the Arduino pins to the Starship functions
#define shift_data 3 // Data for the LED shift registers
#define shift_clk 4 // Clock for the LED shift registers
#define lpod_latch 5 // Register latch for the left Power Pod
#define rpod_latch 6 // Register latch for the right Power Pod
#define disk_left_latch 7 // Latch for the left disk LEDs
#define disk_right_latch 8 // Latch for the right disk LEDs
#define led_red 9 // Red LED for the color sensor
#define led_blue 10 // Blue LED for the color sensor
#define led_green 11 // Green LED for the color sensor
#define disk_speaker 12 // Speaker connection
#define button_input 13 // User button
#define temperature_sensor A0 // Temperature sensor input
#define light_sensor A1 // Light sensor input
#define color_sensor A2 // Color sensor input
#define accl_X A3 // Accelerometer analog X-axis signal
#define accl_Y A4 // Accelerometer analog Y-axis signal
#define accl_Z A5 // Accelerometer analog Z-axis signal

long baseRedBlack = 0; // calibration value for Red LED on Black
target
long baseRedWhite = 0; // calibration value for Red LED on White
target
long baseGreenBlack = 0;
long baseGreenWhite = 0;
long baseBlueBlack = 0;
long baseBlueWhite = 0;

void setup() {
  // Start the Serial connection at 9600 baud
  Serial.begin(9600);

  // Set the pin Modes
```

```

pinMode(led_red, OUTPUT);
pinMode(led_blue, OUTPUT);
pinMode(led_green, OUTPUT);
pinMode(shift_data, OUTPUT);
pinMode(shift_clk, OUTPUT);
pinMode(lpod_latch , OUTPUT);
pinMode(rpod_latch , OUTPUT);
pinMode(disk_left_latch, OUTPUT);
pinMode(disk_right_latch, OUTPUT);
pinMode(disk_speaker, OUTPUT);
pinMode(button_input, INPUT);

// Turn off color sensor LEDs
digitalWrite(led_red, LOW); // Turn off the red LED
digitalWrite(led_blue, LOW); // Turn off the blue LED
digitalWrite(led_green, LOW); // Turn off the green LED

dispOff(); // Turn off all register LEDs

// Calibrate the sensors one time
Serial.println("Ready..place BLACK target and press button");
Serial.println();

while (digitalRead(button_input)) {}

// Get Red value
digitalWrite(led_red, HIGH); // Turn on the red LED

for (int i = 0; i < 20; i++) {
    delay(100);
    baseRedBlack = baseRedBlack + analogRead(color_sensor);
}
baseRedBlack = baseRedBlack / 20;

digitalWrite(led_red, LOW); // Turn off the red LED
digitalWrite(led_green, HIGH); // Turn on the green LED

for (int i = 0; i < 20; i++) {
    delay(100);
    baseGreenBlack = baseGreenBlack + analogRead(color_sensor);
}

```

```

baseGreenBlack = baseGreenBlack / 20;

digitalWrite(led_green, LOW); // Turn on the red LED
digitalWrite(led_blue, HIGH); // Turn on the red LED

for (int i = 0; i < 20; i++) {
    delay(100);
    baseBlueBlack = baseBlueBlack + analogRead(color_sensor);
}
baseBlueBlack = baseBlueBlack / 20;

digitalWrite(led_blue, LOW); // Turn on the red LED

Serial.println("Ready..place WHITE target and press button");
Serial.println();

while (digitalRead(button_input)) {}

// Get Red value
digitalWrite(led_red, HIGH); // Turn on the red LED

for (int i = 0; i < 20; i++) {
    delay(100);
    baseRedWhite = baseRedWhite + analogRead(color_sensor);
}
baseRedWhite = baseRedWhite / 20;

digitalWrite(led_red, LOW); // Turn off the red LED
digitalWrite(led_green, HIGH); // Turn on the green LED

for (int i = 0; i < 20; i++) {
    delay(100);
    baseGreenWhite = baseGreenWhite + analogRead(color_sensor);
}
baseGreenWhite = baseGreenWhite / 20;

digitalWrite(led_green, LOW); // Turn on the red LED
digitalWrite(led_blue, HIGH); // Turn on the red LED

for (int i = 0; i < 20; i++) {
    delay(100);

```

```

    baseBlueWhite = baseBlueWhite + analogRead(color_sensor);
}
baseBlueWhite = baseBlueWhite / 20;

digitalWrite(led_blue, LOW); // Turn on the red LED
}

void loop() {
    long redVal = 0;
    long greenVal = 0;
    long blueVal = 0;

    Serial.println("Ready..press button to test color.");
    Serial.println();

    while (digitalRead(button_input)) {}

    // Get Red value
    digitalWrite(led_red, HIGH); // Turn on the red LED

    for (int i = 0; i < 20; i++) {
        delay(100);
        redVal = redVal + analogRead(color_sensor);
    }
    redVal = redVal / 20;

    digitalWrite(led_red, LOW); // Turn off the red LED
    digitalWrite(led_green, HIGH); // Turn on the green LED

    for (int i = 0; i < 20; i++) {
        delay(100);
        greenVal = greenVal + analogRead(color_sensor);
    }
    greenVal = greenVal / 20;

    digitalWrite(led_green, LOW); // Turn on the red LED
    digitalWrite(led_blue, HIGH); // Turn on the red LED

    for (int i = 0; i < 20; i++) {
        delay(100);
        blueVal = blueVal + analogRead(color_sensor);
    }
}

```

```

    }
    blueVal = blueVal / 20;

    digitalWrite(led_blue, LOW); // Turn on the red LED

    // Calculate the percentages
    redVal = 100 * (redVal - baseRedBlack) / (baseRedWhite -
baseRedBlack);
    greenVal = 100 * (greenVal - baseGreenBlack) / (baseGreenWhite
- baseGreenBlack);
    blueVal = 100 * (blueVal - baseBlueBlack) / (baseBlueWhite -
baseBlueBlack);

    Serial.print("Red = ");
    Serial.print(redVal);
    Serial.println(" %");
    Serial.print("Green = ");
    Serial.print(greenVal);
    Serial.println(" %");
    Serial.print("Blue = ");
    Serial.print(blueVal);
    Serial.println(" %");
    Serial.println();

    if (redVal > greenVal & redVal > blueVal) Serial.println("***
Red...Desert World Found ***");
    else if (greenVal > redVal & greenVal > blueVal)
Serial.println("*** Green...Forest World Found ***");
    else Serial.println("*** Blue...Water World Found ***");
    Serial.println();
}

void dispNum(char ledSet, char ledValue) {
    // set the desired latch low
    digitalWrite(ledSet, LOW);
    // shift out the desired number
    shiftOut(shift_data, shift_clk, MSBFIRST, ledValue);
    // set the desired latch high
    digitalWrite(ledSet, HIGH);
}

```

```
void dispOff() {  
    // Turn all lights off  
    dispNum(rpod_latch, 0);  
    dispNum(lpod_latch, 0);  
    dispNum(disk_left_latch, 0);  
    dispNum(disk_right_latch, 0);  
}
```



## Starship\_lander

This sketch puts you at the controls of Starship to battle gravity as you strive for a soft touchdown before you run out of fuel.

```
// This program simulates the landing of our Starship
// on a planet. The Disk lights show the current velocity, the
// left disk lights are the negative velocity while the right
// lights are positive. The left and right power pods show the
// remaining fuel. Pressing the button fires the decent control
// rockets. Open the Serial Plotter to see the descent in real
// time.
```

```
// Map the Arduino pins to the Starship functions
#define shift_data 3 // Data for the LED shift registers
#define shift_clk 4 // Clock for the LED shift registers
#define lpod_latch 5 // Register latch for the left Power Pod
#define rpod_latch 6 // Register latch for the right Power Pod
#define disk_left_latch 7 // Latch for the left disk LEDs
#define disk_right_latch 8 // Latch for the right disk LEDs
#define led_red 9 // Red LED for the color sensor
#define led_blue 10 // Blue LED for the color sensor
#define led_green 11 // Green LED for the color sensor
#define disk_speaker 12 // Speaker connection
#define button_input 13 // User button
#define temperature_sensor A0 // Temperature sensor input
#define light_sensor A1 // Light sensor input
#define color_sensor A2 // Color sensor input
#define accl_X A3 // Accelerometer analog X-axis signal
#define accl_Y A4 // Accelerometer analog Y-axis signal
#define accl_Z A5 // Accelerometer analog Z-axis signal
```

```
float curFuel = 16.0; // Current Fuel Remaining
float yPos = 10000.0; // Current Altitude
float Accl; // Acceleration terms
float Vel1 = 0.0; // Velocity terms
float Vel2 = 0.0;
const float deltaT = 0.250; // Update time in seconds
```

```
void setup() {
  // Start the Serial connection at 9600 baud
```

```

Serial.begin(9600);

// Set the pin Modes
pinMode(led_red, OUTPUT);
pinMode(led_blue, OUTPUT);
pinMode(led_green, OUTPUT);
pinMode(shift_data, OUTPUT);
pinMode(shift_clk, OUTPUT);
pinMode(lpod_latch , OUTPUT);
pinMode(rpod_latch , OUTPUT);
pinMode(disk_left_latch, OUTPUT);
pinMode(disk_right_latch, OUTPUT);
pinMode(disk_speaker, OUTPUT);
pinMode(button_input, INPUT);

// Turn off color sensor LEDs
digitalWrite(led_red, LOW); // Turn off the red LED
digitalWrite(led_blue, LOW); // Turn off the blue LED
digitalWrite(led_green, LOW); // Turn off the green LED

dispOff(); // Turn off all register LEDs

Serial.println(0); // Set the lower end of the Plotter
}

void loop() {
  // Output current yPos
  Serial.println(yPos);

  // Check for landing
  if (yPos < 1.0) {
    if (Vell > -15.0) {
      // Safe landing
      safeLand();
    } else {
      // Crash
      crashLand();
    }
  }
  while (1) {} // Stop here
}

```

```

    // Calculate next position
    boolean butPush = !digitalRead(button_input) && (curFuel >
0.0);
    if (butPush) curFuel = curFuel - 0.1;
    Acc1 = -9.8 + butPush * 30;
    Vel2 = Acc1 * deltaT + Vel1;
    yPos = Acc1 * (deltaT * deltaT) / 2 + Vel1 * deltaT + yPos;
    Vel1 = Vel2;
    // Display velocity on Disk Lights
    diskVelocity(Vel1);
    fuelLevel(byte(curFuel));

    delay(deltaT * 1000);
}

void dispNum(char ledSet, char ledValue) {
    // set the desired latch low
    digitalWrite(ledSet, LOW);
    // shift out the desired number
    shiftOut(shift_data, shift_clk, MSBFIRST, ledValue);
    // set the desired latch high
    digitalWrite(ledSet, HIGH);
}

void dispOff() {
    // Turn all lights off
    dispNum(rpod_latch, 0);
    dispNum(lpod_latch, 0);
    dispNum(disk_left_latch, 0);
    dispNum(disk_right_latch, 0);
}

void fuelLevel(byte curLevel) {
    byte lpod = 0;
    byte rpod = 0;
    byte vals[9] = {0, 1, 3, 7, 15, 31, 63, 127, 255};
    if (curLevel < 9) {
        lpod = 0;
        rpod = vals[curLevel];
    } else {
        rpod = B11111111;
    }
}

```

```

        lpod = vals[curLevel - 8];
    }
    dispNum(lpod_latch, lpod);
    dispNum(rpod_latch, rpod);
}

void diskVelocity(double curVel) {
    byte vals[9] = {0, 1, 3, 7, 15, 31, 63, 127, 255};
    byte dispV = 0;
    boolean neg = false;
    if (curVel < 0) {
        neg = true;
        curVel = curVel * -1.0;
    }
    for (int i = 1; i < 9; i++) {
        if (curVel >= i * 10.0) dispV = vals[i];
    }
    if (neg) {
        dispNum(disk_left_latch, dispV);
        dispNum(disk_right_latch, 0);
    } else {
        dispNum(disk_right_latch, dispV);
        dispNum(disk_left_latch, 0);
    }
}

void diskSweep(long swpSpeed) {
    int num = B10000000;

    for (int i = 0; i < 9; i++) {
        dispNum(disk_left_latch, num);
        dispNum(disk_right_latch, num);
        num = num / 2;
        delay(swpSpeed);
    }
}

void safeLand() {
    // Turn all lights off
    dispOff();
    delay(500);
}

```

```

    for (int i = 0; i < 17; i++) {
        fuelLevel(i);
        delay(100);
    }
    delay(500);
    diskSweep(50);
    tone(disk_speaker, 500, 200);
    diskSweep(50);
    tone(disk_speaker, 700, 200);
    dispNum(disk_right_latch, 7);
    dispNum(disk_left_latch, 7);
}

void crashLand() {
    // Sad light display
    dispOff();
    delay(500);
    for (int i = 16; i >= 0; i--) {
        fuelLevel(i);
        delay(200);
    }
    delay(500);
    tone(disk_speaker, 400, 500);
    dispNum(disk_left_latch, 192);
    dispNum(disk_right_latch, 192);
}

```

## Starship\_tilt

This sketch uses the Starship as a tilt indicator

```
// This program calibrates the accelerometer and then
// Acts as a tilt sensor to show the relative angle
// in the X and Y axis directions.

// Map the Arduino pins to the Starship functions
#define shift_data 3 // Data for the LED shift registers
#define shift_clk 4 // Clock for the LED shift registers
#define lpod_latch 5 // Register latch for the left Power Pod
#define rpod_latch 6 // Register latch for the right Power Pod
#define disk_left_latch 7 // Latch for the left disk LEDs
#define disk_right_latch 8 // Latch for the right disk LEDs
#define led_red 9 // Red LED for the color sensor
#define led_blue 10 // Blue LED for the color sensor
#define led_green 11 // Green LED for the color sensor
#define disk_speaker 12 // Speaker connection
#define button_input 13 // User button
#define temperature_sensor A0 // Temperature sensor input
#define light_sensor A1 // Light sensor input
#define color_sensor A2 // Color sensor input
#define accl_X A3 // Accelerometer analog X-axis signal
#define accl_Y A4 // Accelerometer analog Y-axis signal
#define accl_Z A5 // Accelerometer analog Z-axis signal

int maxX = 0;
int maxY = 0;
int maxZ = 0;
int minX = 1024;
int minY = 1024;
int minZ = 1024;

void setup() {
  // Start the Serial connection at 9600 baud
  Serial.begin(9600);

  // Set the pin Modes
  pinMode(led_red, OUTPUT);
  pinMode(led_blue, OUTPUT);
```

```

pinMode(led_green, OUTPUT);
pinMode(shift_data, OUTPUT);
pinMode(shift_clk, OUTPUT);
pinMode(lpod_latch , OUTPUT);
pinMode(rpod_latch , OUTPUT);
pinMode(disk_left_latch, OUTPUT);
pinMode(disk_right_latch, OUTPUT);
pinMode(disk_speaker, OUTPUT);
pinMode(button_input, INPUT);

// Turn off color sensor LEDs
digitalWrite(led_red, LOW); // Turn off the red LED
digitalWrite(led_blue, LOW); // Turn off the blue LED
digitalWrite(led_green, LOW); // Turn off the green LED

dispOff(); // Turn off all register LEDs

Serial.begin(9600);
Serial.println("Move the Starship through 360 degrees");
Serial.println("Press the button when done");

int tempAcc;
while (digitalRead(button_input)) {
    tempAcc = analogRead(accl_X);
    if (tempAcc > maxX) maxX = tempAcc;
    if (tempAcc < minX) minX = tempAcc;
    tempAcc = analogRead(accl_Y);
    if (tempAcc > maxY) maxY = tempAcc;
    if (tempAcc < minY) minY = tempAcc;
    tempAcc = analogRead(accl_Z);
    if (tempAcc > maxZ) maxZ = tempAcc;
    if (tempAcc < minZ) minZ = tempAcc;
}

Serial.println("Calibration Complete.");
}

void loop() {
    int curVal = analogRead(accl_X);
    acclLevelX(curVal);
    curVal = analogRead(accl_Y);

```

```

    acclLevelY(curVal);
    delay(100);
}

void dispNum(char ledSet, char ledValue) {
    // set the desired latch low
    digitalWrite(ledSet, LOW);
    // shift out the desired number
    shiftOut(shift_data, shift_clk, MSBFIRST, ledValue);
    // set the desired latch high
    digitalWrite(ledSet, HIGH);
}

void dispOff() {
    // Turn all lights off
    dispNum(rpod_latch, 0);
    dispNum(lpod_latch, 0);
    dispNum(disk_left_latch, 0);
    dispNum(disk_right_latch, 0);
}

void acclLevelX(int curLevel) {
    byte vals[9] = {0, 1, 2, 4, 8, 16, 32, 64, 128};
    if (curLevel > maxX) curLevel = maxX;
    if (curLevel < minX) curLevel = minX;
    int diffX = (curLevel - minX) / ((maxX - minX) / 8);
    dispNum(lpod_latch, vals[diffX]);
    dispNum(rpod_latch, vals[diffX]);
}

void acclLevelY(int curLevel) {
    byte vals[9] = {0, 1, 2, 4, 8, 16, 32, 64, 128};
    if (curLevel > maxX) curLevel = maxX;
    if (curLevel < minX) curLevel = minX;
    int diffY = (curLevel - minX) / ((maxX - minX) / 16);
    dispNum(disk_left_latch, 0);
    dispNum(disk_right_latch, 0);
    if (diffY < 9) dispNum(disk_right_latch, vals[diffY]);
    else dispNum(disk_left_latch, vals[17 - diffY]);
}

```



## Starship\_alarm

This sketch sounds an alarm if the Starship feels an impact.

```
// Sound the alarm if the Ship is hit

// Map the Arduino pins to the Starship functions
#define shift_data 3 // Data for the LED shift registers
#define shift_clk 4 // Clock for the LED shift registers
#define lpod_latch 5 // Register latch for the left Power Pod
#define rpod_latch 6 // Register latch for the right Power Pod
#define disk_left_latch 7 // Latch for the left disk LEDs
#define disk_right_latch 8 // Latch for the right disk LEDs
#define led_red 9 // Red LED for the color sensor
#define led_blue 10 // Blue LED for the color sensor
#define led_green 11 // Green LED for the color sensor
#define disk_speaker 12 // Speaker connection
#define button_input 13 // User button
#define temperature_sensor A0 // Temperature sensor input
#define light_sensor A1 // Light sensor input
#define color_sensor A2 // Color sensor input
#define accl_X A3 // Accelerometer analog X-axis signal
#define accl_Y A4 // Accelerometer analog Y-axis signal
#define accl_Z A5 // Accelerometer analog Z-axis signal

int sens = 5;

void setup() {
  // Start the Serial connection at 9600 baud
  Serial.begin(9600);

  // Set the pin Modes
  pinMode(led_red, OUTPUT);
  pinMode(led_blue, OUTPUT);
  pinMode(led_green, OUTPUT);
  pinMode(shift_data, OUTPUT);
  pinMode(shift_clk, OUTPUT);
  pinMode(lpod_latch, OUTPUT);
  pinMode(rpod_latch, OUTPUT);
  pinMode(disk_left_latch, OUTPUT);
  pinMode(disk_right_latch, OUTPUT);
```

```

pinMode(disk_speaker, OUTPUT);
pinMode(button_input, INPUT);

// Turn off color sensor LEDs
digitalWrite(led_red, LOW); // Turn off the red LED
digitalWrite(led_blue, LOW); // Turn off the blue LED
digitalWrite(led_green, LOW); // Turn off the green LED

dispOff(); // Turn off all register LEDs
}

void loop() {
  while (digitalRead(button_input)) {}
  for (int i = 0; i < 7; i++) {
    dispNum(lpod_latch, B00111111 >> i);
    delay(800);
  }
  int curX = analogRead(accl_X);
  int curY = analogRead(accl_Y);
  int curZ = analogRead(accl_Z);

  boolean alarm = false;

  while (!alarm) {
    if (analogRead(accl_X) > curX + sens) alarm = true;
    if (analogRead(accl_Y) > curY + sens) alarm = true;
    if (analogRead(accl_Z) > curZ + sens) alarm = true;
    if (analogRead(accl_X) < curX - sens) alarm = true;
    if (analogRead(accl_Y) < curY - sens) alarm = true;
    if (analogRead(accl_Z) < curZ - sens) alarm = true;
  }
  alarmAnim();
}

void alarmAnim() {
  for (int t = 0; t < 3; t++) {
    dispNum(disk_left_latch, 192);
    dispNum(disk_right_latch, 192);
    for (int i = 8; i >= 0; i--) {
      tone(disk_speaker, 600 - i * 20, 25);
      delay(25);
    }
  }
}

```

```

    }
    dispNum(disk_left_latch, 0);
    dispNum(disk_right_latch, 0);
    delay(200);
}
}

void dispNum(char ledSet, char ledValue) {
    // set the desired latch low
    digitalWrite(ledSet, LOW);
    // shift out the desired number
    shiftOut(shift_data, shift_clk, MSBFIRST, ledValue);
    // set the desired latch high
    digitalWrite(ledSet, HIGH);
}

void dispOff() {
    // Turn all lights off
    dispNum(rpod_latch, 0);
    dispNum(lpod_latch, 0);
    dispNum(disk_left_latch, 0);
    dispNum(disk_right_latch, 0);
}

```