# HappyLab

# MIT 6.270 IAP:2012

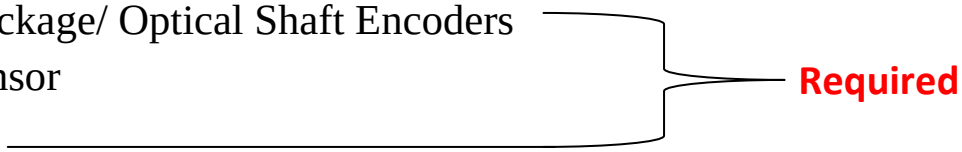
"The devil is in the details." Thus, the Purpose of the HappyLab is to clear up those tiny details by going over step by step in wiring and programming all of the sensors and motors used in 6.270. The document will be picture heavy and will try to be as concise and to the point as much as possible.

Steven Jorgensen
6.270 Organizers
**Massachusetts Institute of Technology**
January 2012

## 6.270 Happy Lab

### Table of Contents



# 6.270 Happy Lab

## Step 0.0 Setting up a Programming Environment

**The following instructions can also be viewed at the wiki page in the 6.270 website:** https://scripts-cert.mit.edu:444/~6.270/wiki/index.php?title=Setting_Up_A_Development_Environment

**Windows Computer**

**Step 1**: **Install USB Drivers**

Go to http://www.ftdichip.com/Drivers/VCP.htm and download the "VCP driver" for your operating system. Then install the software, following the appropriate install guide. (Which you can download here (http://www.ftdichip.com/Support/Documents/InstallGuides.htm) You'll need to have a HappyBoard handy.

**Step 2: Get AVR Tools**

Install WinAVR (http://winavr.sourceforge.net/) This application bundle comes with Programmer's Notepad

(Start -> All Programs -> WinAVR -> Programmer's Notepad), which you can use to edit C files.

***Note: This is where you will be typing and uploading your code.***

**Step 3: Install GIT**

Install *msysgit* from http://msysgit.googlecode.com/files/Git-1.7.4-preview20110204.exe ,

**Step 4: Get JoyOS**

You will have to use Git Bash to download JoyOS. Please read Scott's **Git Introduction for 6.270** under the Course Information on the main website to download get JoyOS from the Athena clusters.

**Important Note:** It is incredibly important that you read through the tutorial completely! Cutting Corners will cause you to take longer.

---

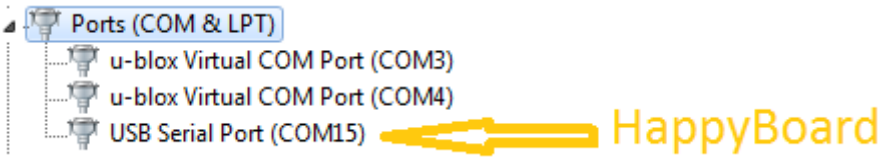
**Step 5: Obtain the Happyboard's Communication Port**

# 6.270 Happy Lab

On Windows the Happyboard will show up as a serial port named **COM***. The correct COM port number can be determined in the Device Manager.

Right click on ”My Computer”, click ”Properties”, then ”Hardware”, then ”Device Manager”. Under the ”Ports (COM & LPT)” section, the last COM port should be the Happyboard. Alternatively, **if you’re using windows 7, just type “device manager” on the quick search.**

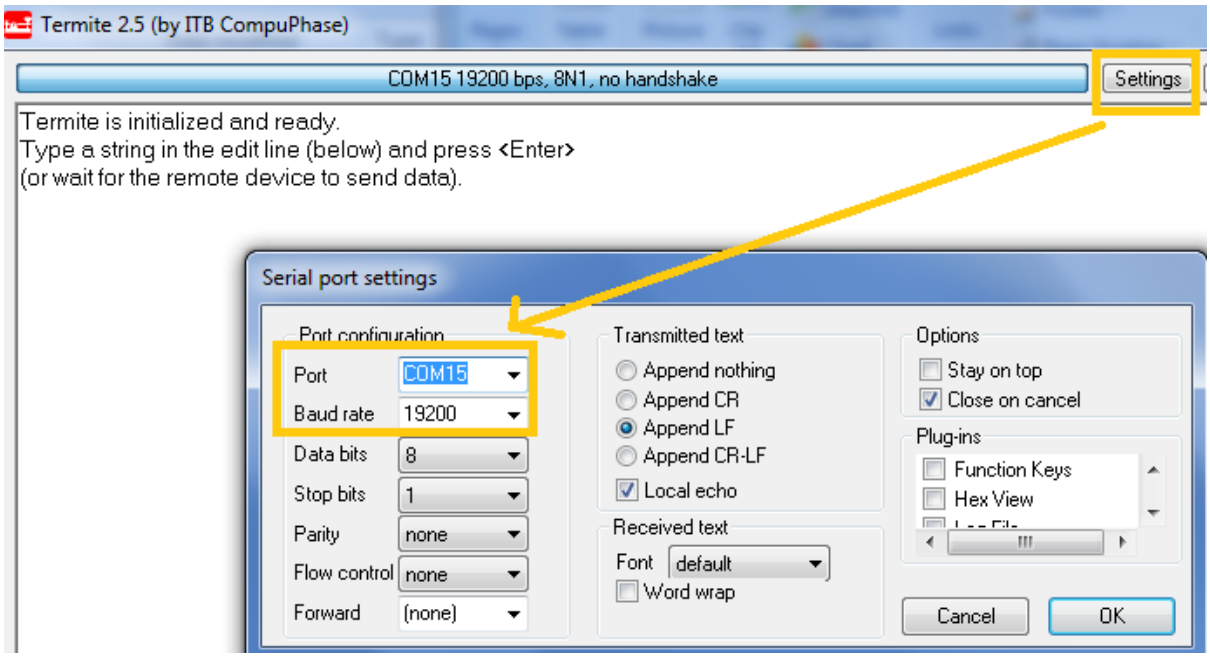
In my Windows 7 Device, this is how happyboard shows up:




**Step 6: Install a Serial Terminal**

Get Termite (http://www.compuphase.com/software_termite.htm)

Open Termite, Under Settings change **Port** to the port you found above, **Baud rate** is 19200.

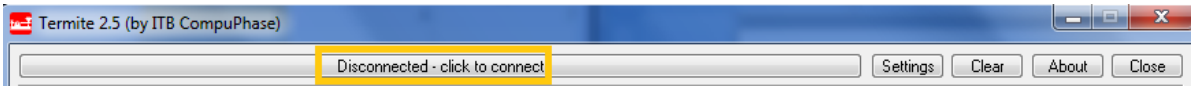



*The serial terminal is the way you'll talk to the HappyBoard. When you print text or read user input, it will be through a serial terminal.*

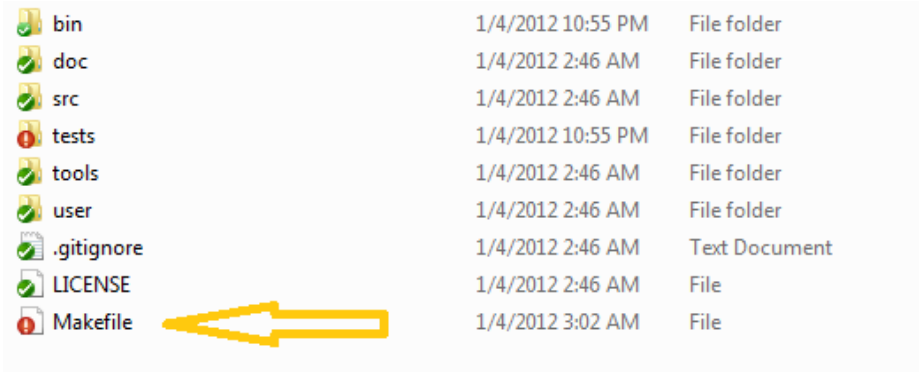
***Important Note:** In general, only one program at a time can have exclusive rights to talk to the happyboard. This means you must close any serial terminals before attempting to download a new program!*

**Step 7: Building and Uploading a Program**

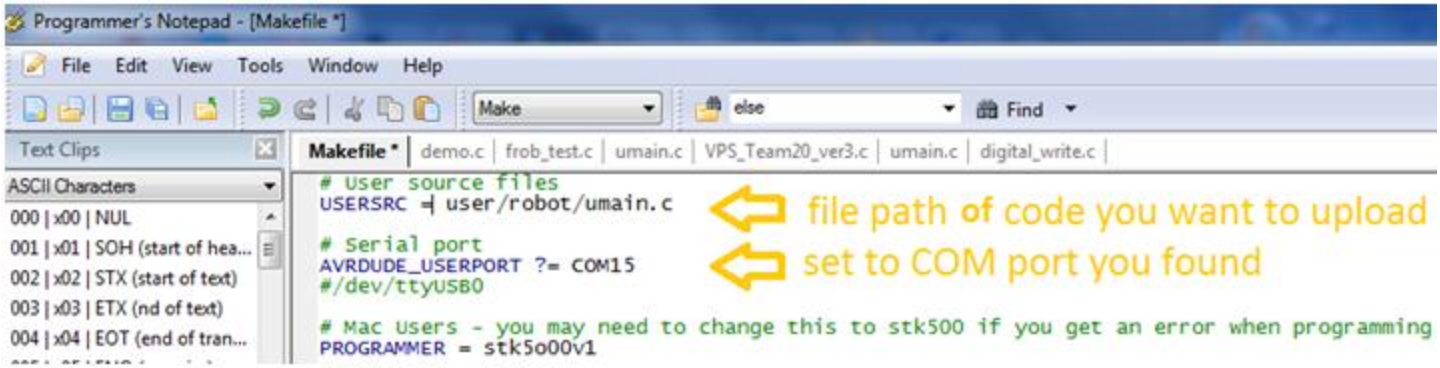
1) **Connect the HappyBoard to the USB**
2) **Press and Hold the STOP Button on the happyboard and turn on the happyboard. Wait until Both the red and green LEDs are solid. Let go of the STOP button after 3 seconds.**
3) **Make sure your serial terminal is disconnected!!**



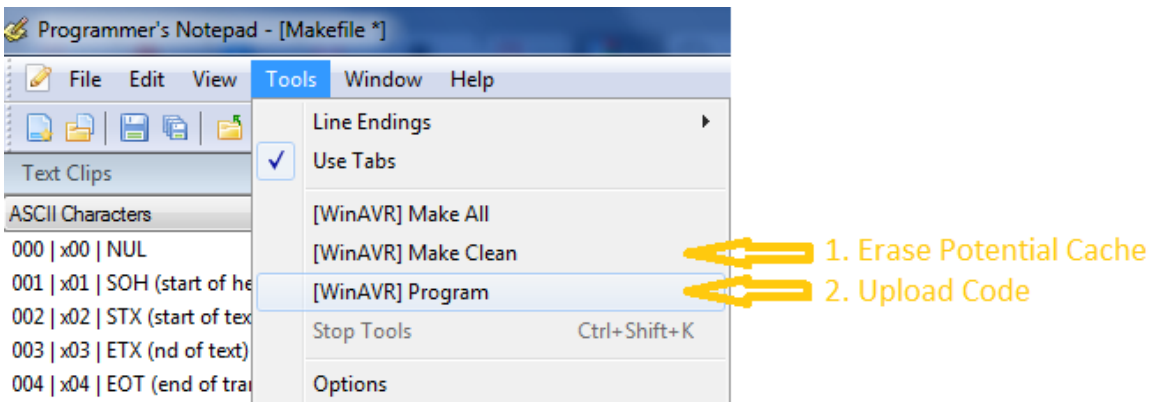

4) **Navigate to the JoyOS Directory you downloaded with git.**
5) **Open Makefile using Programmer's Notepad**




6) **Make the Necessary changes under USERSRC and AVRDUDE_USERPORT**

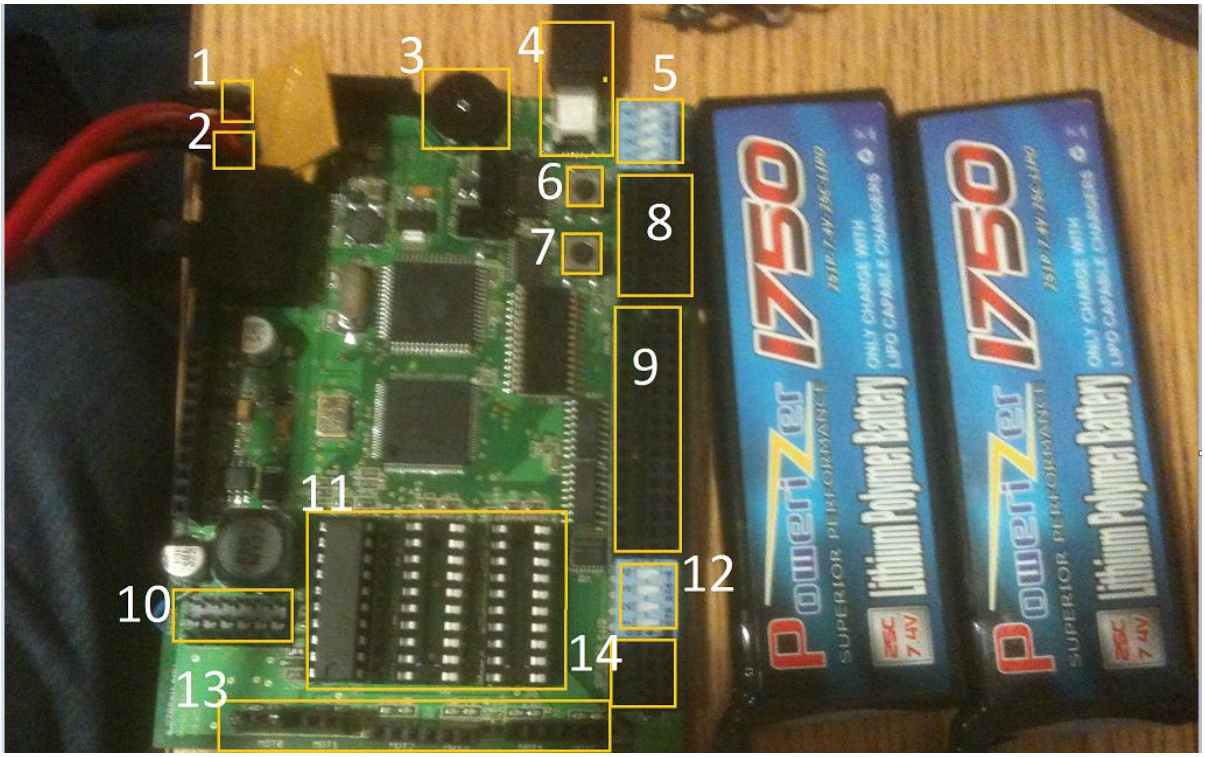



7) **Press Tools -> Make Clean -> Make Program to upload your code. Always upload code in this order.**



# 6.270 Happy Lab

## Step 1.0 Intro to HappyBoard




This year's happyboard now have 2 battery packs, one for logic and one for motor power. Other new features have to do with the *firmware*, software, and extra hardware on the board. I'll describe them below

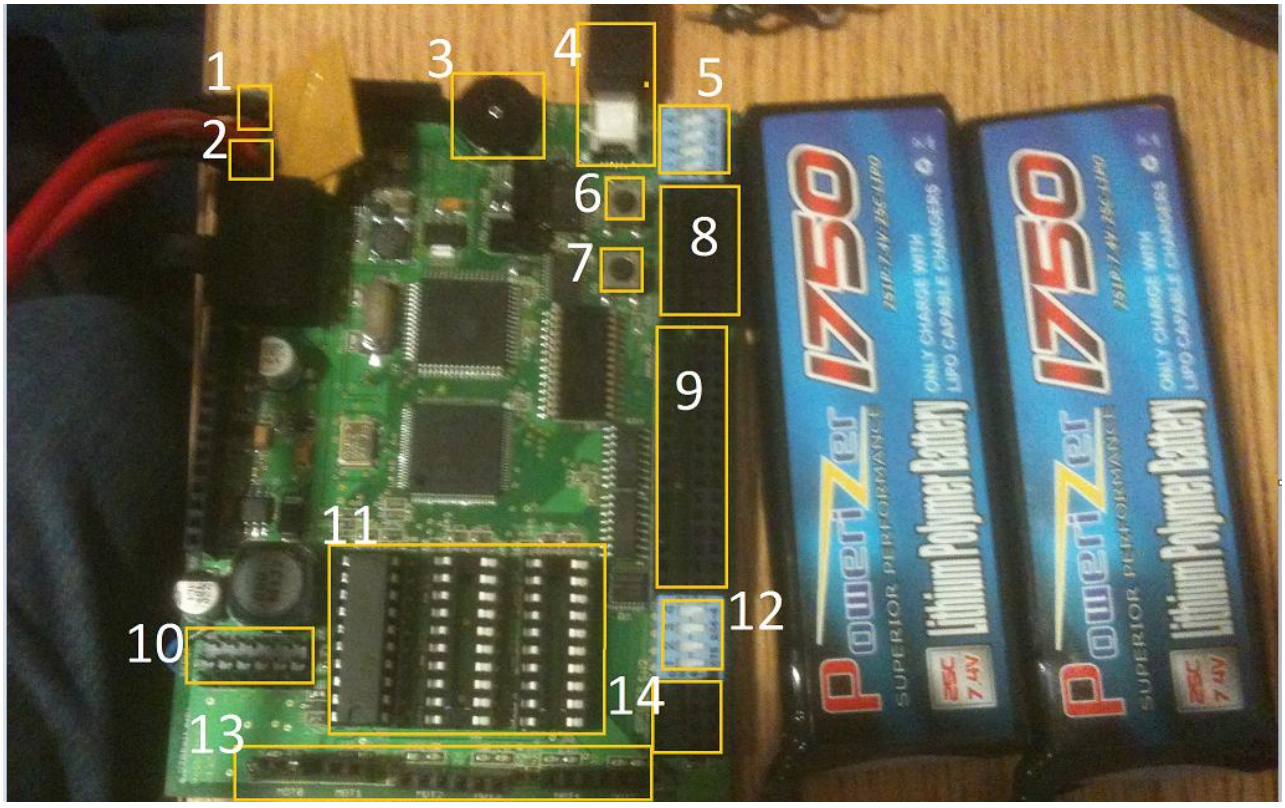
1) **Logic Power Supply –** Small *LiPo battery* to power the happyboard's firmware.
2) **Motor Power Supply –** Big LiPo battery to power motors and Servos.

   **Note:** These are LiPo batteries. They are known for having a very high power density, which means they are potentially dangerous if mishandled. Make sure you keep an eye on them while charging, and never leave them charging unattended or overnight!

   **Peter's Note:** Batteries may become warm during charging or normal operation, but if they swell or become very hot to the touch, they should be disconnected and a staff member should be notified. Do not mess with faulty batteries. Be aware that dangling charging connectors are wired directly to the battery cells, and if they brush against exposed metal objects like the pins of the servo headers, they can short the battery and burn out your HappyBoard in a puff of smoke (true story).
3) **Frob Knob** – also known as a *potentiometer*! A new hardware modification is that this potentiometer locks at the center. We'll tell you how to use this in the HappyLab.
4) **Mini USB Port**. The other USB end of this cable goes to your computer.
5) **Digital Input Pull Up Resistors –** Adds a *pull-up resistor* which provides 5V from Signal to GND for the first 4 sensors on the Digital Input pins. (Pins 0-3).Always keep these switches to ON.
6) **STOP button.** Essential for the programming routine
7) **GO button.** Essential for the start-up routine unless otherwise specified by the main server.
8) **Digital Inputs/Outputs -** only digital sensors can go in these slots (eg: limit switches, push buttons) From Left to right the pin slots are SIGNAL, POWER, and GND. In addition, these pins can act as Digital Outputs of HIGH or

LOW. This is useful if you want to use LEDs to indicate status or do testing. Spans from 0-7. ***All of the pins have a pull-up resistor. By default, pins 0-3 have their resistors connected.***

***9)*** **Analog Inputs.** Analog Inputs go here (eg: Gyroscopes, Sharp Distance Sensor). Although Digital Inputs can be plugged in these ports, it is highly recommended that you separate the inputs. Spans from 8 to 23. ***All of the pins have a pull-up resistor. By default, pins 20-23 have their resistors connected (Pull ups are switch ON).***




10) **Servo Inputs.** NOTE! This a SERVO motor is different from a DC Motor in this case. Servo motors in 6.270 are positional servos with a range from 0-180 degrees. However, you can also make them continuous by breaking the mechanical stop inside. Regardless, only servos connect here. **You can connect 6 Servos**. **From top to bottom, the pins are GND, PWR, SIG.**
11) **Motor Drivers –** These control the speed of the DC motor. Each Driver can control 2 motors, and the happyboard supports up to 3 drivers. In the picture above, I only have 1 motor driver so I can only control 2 motors.
12) **Pull Up Resistors –** Adds a pull up resistor from Signal to GND for the last 4 inputs on the Analog Input pins. (Pins 20-23). By default, they are all switched to ON.
**13)** **DC Motor Pins.** You can only connect 6 DC motors to the happyboard from MOT01 to MOT05. For each motor connection there are 3 pins. Only 2 of these matters. From Left to right it is POS, n/a, NEG.

> **IMPORTANT: In this case only,** POS and NEG are interchangeable your motors would spin the other way if it the connection is inverted. **IT IS INCREDIBLY IMPORTANT YOU GET A SYSTEM FOR FIGURING OUT WHICH PIN GOES WHERE.** *Matches have been lost due to lack of convention.*

**14)** **Encoder inputs –** These are high-speed digital inputs that can be used with rotational shaft encoders to measure motor speeds and distances. **From left to right, Pin Slots are still SIGNAL, POWER, GND.** Spans from 24-27

# 6.270 Happy Lab

## Step 2.0 Uploading Sample Programs

Now we will try to make sure you can navigate JoyOS and upload a sample program.

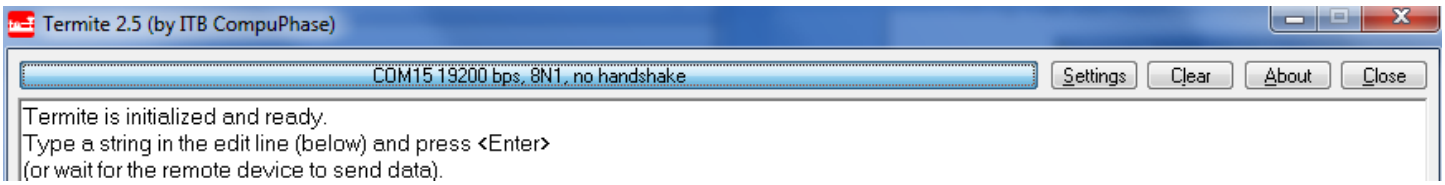
1) Open frob_test.c under the *tests* folder using Programmer's notebook(or your choice of development tool).
2) Try to read the code and see if you can understand what it's trying to do. It's ok if you can't.
3) Save the file.
4) Open Makefile
5) Change **USERSRC = user/robot/umain.c**
6) To **USERSRC = tests/frob_test.c**

   ***Note: This shows how you can use the tests files within JoyOS to check other code snippets. Hopefully you realize that you don't have to upload from user/robot/umain.c***
7) Save Makefile
8) Upload the code using the appropriate steps depending on which operating system you are using.

```
avrdude.exe: input file bin/libjoyos.hex auto detected as Intel Hex
avrdude.exe: writing flash (26782 bytes):

Writing | ################################################## | 100% 18.37s

avrdude.exe: 26782 bytes of flash written

avrdude.exe done.  Thank you.


> Process Exit Code: 0
> Time Taken: 00:20
```

9) After Uploading the code. Turn off your happyboard

Open your Serial Terminal. Make sure it is connected to the happy board. If you're using termite, it should look:




10) Turn on your Happyboard. It should will run usetup(void) for the start up routine. Wait until it tells you to press the GO button on your happyboard.
11) By now, numerical print out of the **frob knob's value** should be visible. Note: This is super helpful for figuring out gains, empirical settings, and other calibration needed for your robot.

```
frob[0,100] = 22  (228)
frob[0,100] = 22  (227)
frob[0,100] = 23  (240)
frob[0,100] = 29  (298)
frob[0,100] = 36  (374)
frob[0,100] = 43  (442)
frob[0,100] = 51  (522)
```

12) Change **USERSRC = tests/frob_test.c** back to **USERSRC = user/robot/umain.c**

**IF you made it this far, CONGRATS! You're ready to spend the next month trying to program a winning robot!**

**IF NOT! Be sure to find a 6.270 organizer who can help you out.**

# 6.270 Happy Lab

## Step 3.0 HappyTest

# <STOP>

Before proceeding, you need to know that there's a special way to test all of your motors and sensors. It's called happy test. For all of the sensors we will be doing, **except for the gyroscope and the IR Sharp Distance Sensor**, you can simply upload one program to test them all.

Navigate to **user/happytest/happytest.c**

**Follow Step 2.0 Uploading Sample Programs** to upload happytest.c appropriately to your happyboard.

Open your serial terminal and you'll see that you can test Servos, Motors, Digital Inputs, Analog Inputs, and Encoder Inputs.

```
Happytest v0.61
Servo Test: press Go (or Stop to skip)
Motor Test: press Go (or Stop to skip)
Digital Test: press Go (or Stop to skip)
Analog Test: press Go (or Stop to skip)
Encoder Test: press Go (or Stop to skip)
panic: Testing new panic()
```

**Commands:**
Run a test by pressing GO.
Skip a test by Pressing STOP
End a test by Pressing STOP
Frob Knob- Changes Analog/Digial Inputs to explore & the Motor and Servo Outputs

Just try running it and appreciate its beauty.

# </STOP>

# 6.270 Happy Lab

## Step 4.0 Motors and Sensors

**IMPORTANT NOTE:** If you don't know how to solder or strip a wire, ask to be taught by any of the organizers.

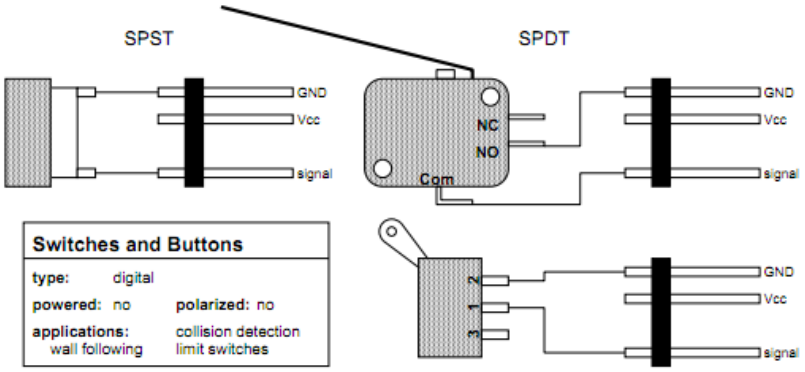
### 4.1 Limit Switches and Buttons

**Background info**: Switches and Buttons are perhaps the most intuitive sensors. They are digital which means they only have a binary state (1 or 0). In 6.270, you'll most likely encounter two types of switches: SPST (Single Pole Single Throw) and SPDT(Single Pole Double Throw)

SPST – Only has one terminal. If it is activated, the circuit is complete or CLOSE , otherwise it is disconnected or OPEN.

SPDT – Has 3 terminals. Common("C"), Normally Closed (NC) and Normally Open (NO). When the switch is activated, Common is connected to Normally Open, otherwise it is connected to Normally Closed. SPDT can be treated as a SPST if the Normally Closed terminal is ignored as shown below:

**Wiring Diagram**




After doing the necessary wiring, remember that this is a Digital Input. Which means it can be plugged in anywhere from pin slot 0 to pin slot 23. It is recommended however, that you plug it in pins 0-7.

**Programming a Limit Switch:** Luckily, it's easy to program a limit switch. Since it is an input, all we have to do is read it using the command digital_read(#PIN_number). Here's a sample Snippet. Try running this using your serial terminal:

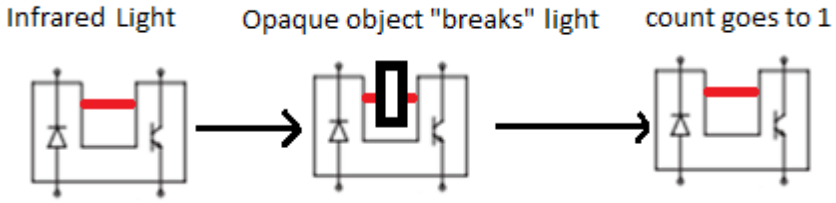
**Open user/robot/umain.c** Try entering this piece of code under umain(void):

```
// Entry point to contestant code.
int umain (void) {
   // YOUR CODE GOES HERE
    while (1) {
        if (digital_read(0) == 1){
            printf("TRUE!\n");
        }
        else{
            printf("FALSE!\n");
        }
    }
    // Will never return, but the compiler complains without a return
    // statement.
    return 0;
}
```

Remember to have your **pullup switched to ON, on the digital inputs**.

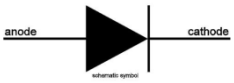
## 4.2 BreakBeam Sensor Package/Optical Shaft Encoders

**IMPORTANT NOTE:** Please be careful when wiring the optical shaft encoders. There are 4 wires and a 330Ω resistor connected. In addition, it can only be connected on pin slots 24-27.



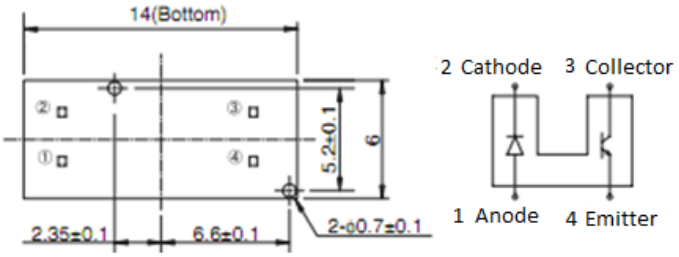

**Background info**: The breakbeam sensor has an infrared LED (Light-Emitting-Diode) and a Phototransistor that is sensitive to light. When an opaque/solid object blocks the path of the light, the light beam "breaks" and the resistance value changes. This becomes very useful when you want to keep track of how many times the light has been broken. You can then keep track of how many times a wheel has turned.

**Wiring Diagram:** In electronic schematics, a Diode is depicted as:



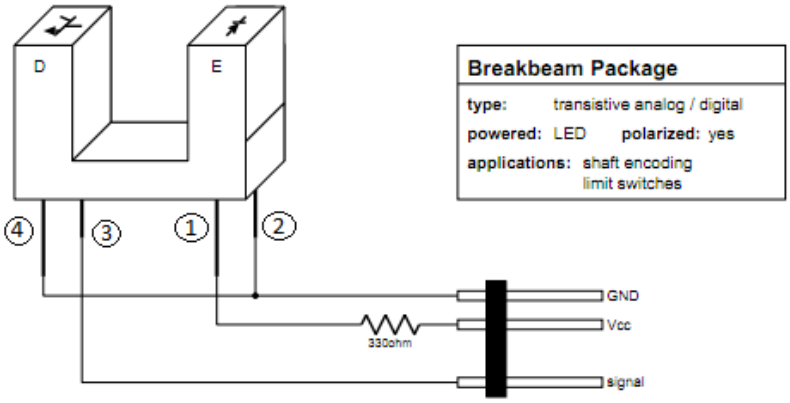

Locate the **Diode** Symbol. Look at it from the bottom and make sure it is oriented as specified below:

**Bottom View**




The diagram below clearly depicts where to solder each pins. Be very careful and don't forget the 330ohm resistor on the LED side. "E" is for emitter and "D" is for detector. A nice way of thinking where the wires go is that the **GNDs are diagonal from each other.**

**Back View:**



# 6.270 Happy Lab

**Programming shaft encoders**

***On your happyboard remember that encoders can only go to pins 24-27.***

Luckily for you, there is only one simple command that brings up the value of the encoder and it is called encoder_read(PIN_NUMBER). Calling this function returns an integer value. In my case I used pin number 25.

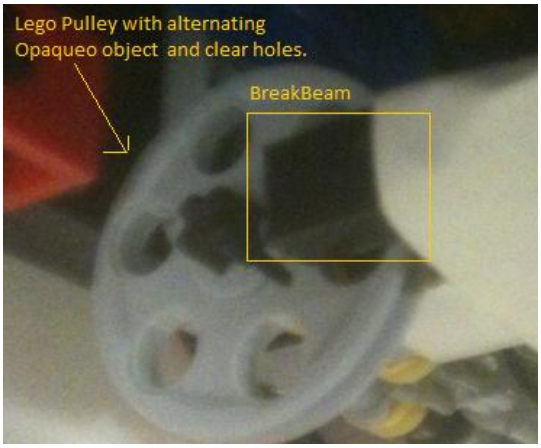
Use **happytest** to try see if your encoder works or alternatively, try running your own snippet

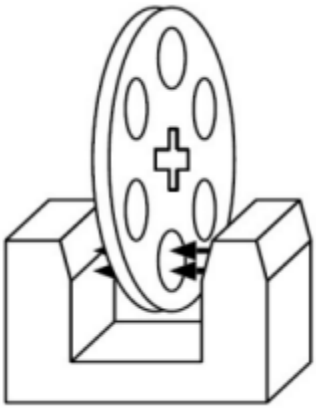
**Open user/umain.c** Try entering this piece of code under umain(void):

```
// Entry point to contestant code.
int umain (void) {
    // YOUR CODE GOES HERE
    while (1) {

        printf("Encoder Value: %u\n", encoder_read(25));

    }
    // Will never return, but the compiler complains without a return
    // statement.
    return 0;
}
```

To reset the encoder value to 0, use the void command **encoder_reset(PIN_NUMBER);**

Run your serial terminal and try repeatedly breaking the infrared light with an opaque object. Preferably, use a *lego pulley* as shown below:



If you wired everything appropriately and entered the code, it should print something very similar to:

```
Encoder Value: 8
Encoder Value: 8
Encoder Value: 8
Encoder Value: 9
Encoder Value: 9
Encoder Value: 9
```

**Important Note:** Notice how the encoder only *knows how to count up.* There is no sense of direction for a simple break beam encoder. If you absolutely need to measure the number of rotations AND direction, there is a way to do this using 2 break beam sensors. It's still experimental though, so talk to a TA for more info.

## 4.3 GyroScope Sensor

**Background Info:** A Gyroscope is an analog sensor that figures out the angular position of your robot. A Gyroscope is an analog sensor that measures how fast your robot turns. Why do you care how fast your robot is turning? If you integrate the rotational velocity with respect to time (radians per second), you end up with the angle your robot is heading (radians), which is very important for navigation! The Happyboard handles the integration for you, so you can think of the gyro as measuring the robot's heading.

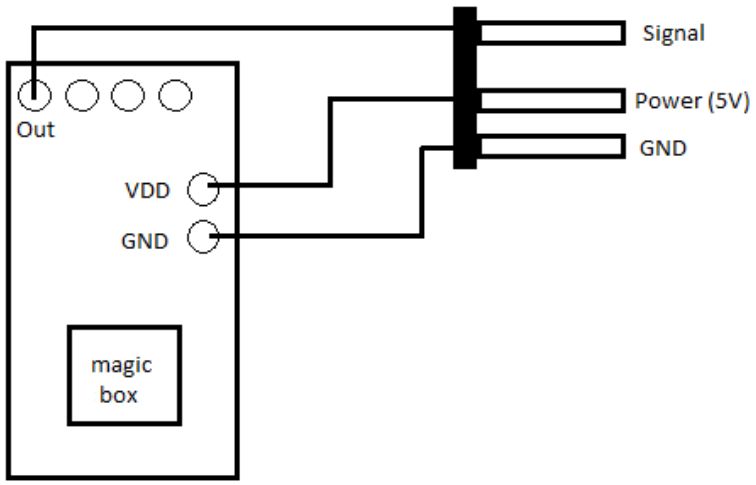
Important Limitation: Since we integrate the gyroscope's measurement, any small amount of noise in the signal adds up, which means the value returned by gyro_get_degrees() will slowly drift over time.

Being analog, this means it can only go from pins 8-23. Gyroscopes only measure the rotation about an axis perpendicular to the plane of the gyro board. There's an entire description of how gyroscopes work using the course notes on the class website. For practicality reasons, there's no need to know how the magic black box works.

**Important Note:** The gyroscope itself doesn't necessarily have to be connected on your happyboard. It can be placed anywhere on the robot but remember that the gyroscope will only measure rotation about an axis perpendicular the gyroboard.

**Wiring a GyroScope:**

There are only 3 terminals needed to wire a gyroscope: Output, Power, and Ground.




Remember that the Gyroscope is an analog input so it can only connect from pins 8 to 23. We recommend using pin 20,21, 22 or 23 with the pull-up resistor switched OFF for that pin.

# 6.270 Happy Lab

**Programming a GyroScope:**

It's probably easier to navigate to **user/gyrotest/umain.c** to get the sample code for a gyroscope.

```
#include <joyos.h>

#define GYRO_PORT       11
#define LSB_US_PER_DEG  1400000
```

1

```
// usetup is called during the calibration period. It must return before the
// period ends.
int usetup (void) {
    printf("\nPlace robot,    press go.");
    go_click ();
    printf ("\nStabilizing...");
    pause (500);
    printf ("\nCalibrating     offset...\n");
    gyro_init (GYRO_PORT, LSB_US_PER_DEG, 500L);
    return 0;
}

int umain(void) {
    for (;;) {
        printf ("\ntheta = %.2f", gyro_get_degrees());
        pause (100);
    }
    return 0;
}
```

2

3

The boxes above show the essential code you need to get the value of the gyroscope output. Everything else is sugar

*Box 1*: These are *constants.* It is good programming practice to name all the constants you need. This significantly cleans your code. **GYRO_PORT** specifies where the gyroscope is plugged. **LSB_US_PER_DEG** (shorthand for least-significant-bit microseconds per degree*)* is essentially a constant conversion factor. This needs to be changed slightly for every gyroscope.

*Box 2*: gyro_init(Port#, Conversion_Constant, MiliSecond_Calibration_Time) The last argument of gyro_init tells happyboard how long must it sample the nominal voltage value of the gyroscope. 500ms is good enough. More than 1000ms is probably overkill and unnecessary.

*BOX 3*: gyro_get_degrees() returns the degree heading where counterclockwise is positive. The value continues to increase even after a complete turn eg: After 2 complete turns, gyro_get_degrees() will return 720. When gyro_init is running (e.g. for 500ms), make sure not to move your robot, or it will mess up the calibration.

Run the code and look at your serial terminal. It should print out the Theta values of the gyroscope. Try rotating the robot in place:

```
theta = -0.22
theta = 1.38
theta = 2.35
theta = 3.01
theta = 3.60
theta = 3.75
```

# 6.270 Happy Lab

**Important Note:** You might notice that even though the gyroscope is stationary, the values increase/decrease over time. This is called **"drifting."** It is important to then calibrate our gyroscope.

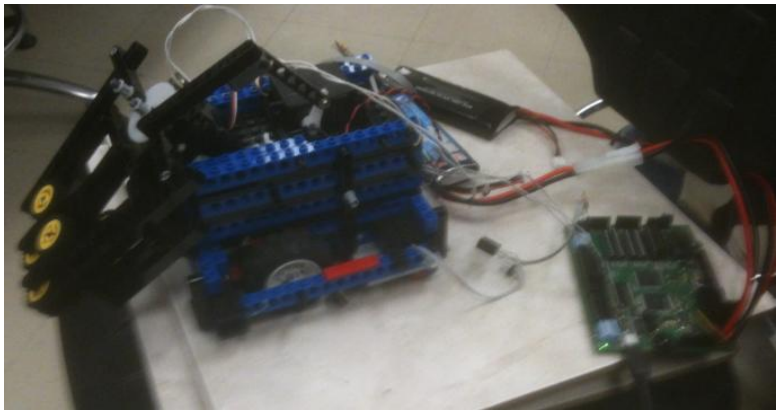
**Calibrating a GyroScope**

Step 1:Upload the gyrotest code from user/gyrotest/umain.c

Step 2: Place the Robot on a rotating platform/chair. **Ensure that the gyroscope is parallel to the ground**. This is very important.

*The chairs in the lab are slanted so feel free to use this setup to calibrate your robot. I used an Android App to level it. You can use traditional Levels or a smart phone app.*

Step 3: Take note of your starting position. Turn on the robot while it is connected to your computer. Open the serial terminal and make sure the robot is in its nominal value (Make sure it's printing out 0 Degrees or it rounds to 0.)



Step 4: Disconnect the MiniUSB cable and spin the platform 10 times. Plug the USB cord back in to your computer. If it doesn't update, disconnect and reconnect using the terminal. If you can't get this to work, Just have the USB cable plugged in the entire time. Careful as the wire tends to gets tangled to itself.

This is what I got:

# 6.270 Happy Lab

theta = -3735.39
theta = -3735.35
theta = -3735.31
theta = -3735.24
theta = -3735.22
theta = -3735.20
theta = -3735.19

Step 5: Use the following ratio to figure out your new LBS_US_PER_DEG constant:

$$\frac{1400000}{3600} = \frac{x}{Experimental\ Value\ in\ Degrees}$$

Solve for x: $x = \frac{1400000 * experimental\_value}{3600}$

In my case I get x = 1452500

Step 6: Use this new value and repeat the experiment. See if you get closer to 3600 after 10 turns.

After 10 turns, I got 3605. Theoretically it should have been 3600. We have an error of 5 degrees now. This is pretty good because before, we were off by 135 degrees.

Before calibration our error was $\frac{(3735-3600)}{3600} * 100 = 3.75\%$
After Calibration our new error is $\frac{(3605-3600)}{3600} * 100 = 0.14\%$

**Important Note:** Calibration of the LSB_US_PER_DEG constant has nothing to do with drift. It is just calibrating the multiplier of the rotation angle. Let's say your LSB_US_PER_DEG is too small. When you turn your robot 180 degrees, you'll get some larger measured value, say 200 degrees. Now if you turn your robot back 180 degrees the opposite direction, you'll end up back at 0. But if you had continued 180 degrees in the same direction, you would have gotten a reading of 400 degrees instead of 360! This isn't time-induced drift (which is caused by integration), but rather rotation-induced drift: the more you rotate, the worse your angles get

**Supplemental:** Try also rotating it 10 times in one direction, and 10 times back. See how far it is off from 0.

# 6.270 Happy Lab

## 4.4 Wiring and Programming a DC Motor

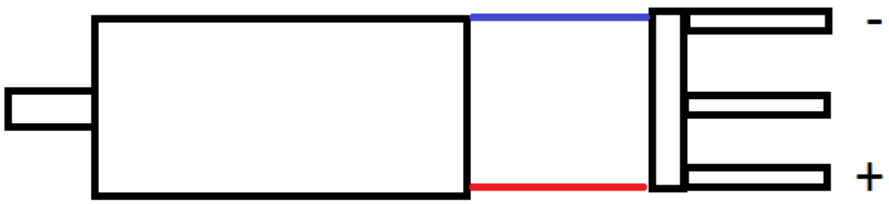
**Background Info:**

DC motor is short for (Direct Current Motor). The 6.270 motors used this year have a high **RPM** (Revolutions Per Minute) and a low **Torque.** For DC Motors speed and torque are inversely related to one another. The higher the RPM is, the lower the torque available and vice versa. This tradeoff is usually addressed using gear reductions (Low RPM, High Torque or high RPM and LOW torque). Since your robot will require high-torque tasks and the DC motors have room to trade off RPM, it is advisable to use a gearbox to find the proper balance.

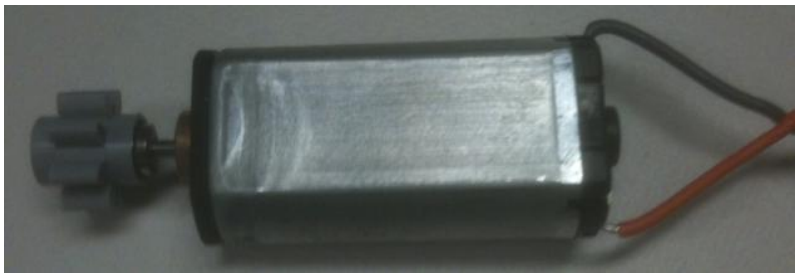
The speed of the motor output can also be controlled using a method called **PWM** (Pulse-Width-Modulation). In essence, a motor will be switched ON and OFF over a period of time. By limiting the amount of current going through the motor, you can then control the motor's **duty cycle**(how long a motor is turned ON over a set period of time). For example, operating the motor at **50% duty cycle** means that the motor will have 50% of its original RPM and 50% of its original torque output. Take note that because we are essentially decreasing the voltage input by 50%, we are only decreasing the motor's overall capability by 50% to get a desired speed. Changing RPM using PWM does not magically grant extra torque. You can think of PWM as a scaling factor of RPM and Torque.

**Wiring the Motor**

Motors are wired very simply as follows:




Each terminal gets its own wire.

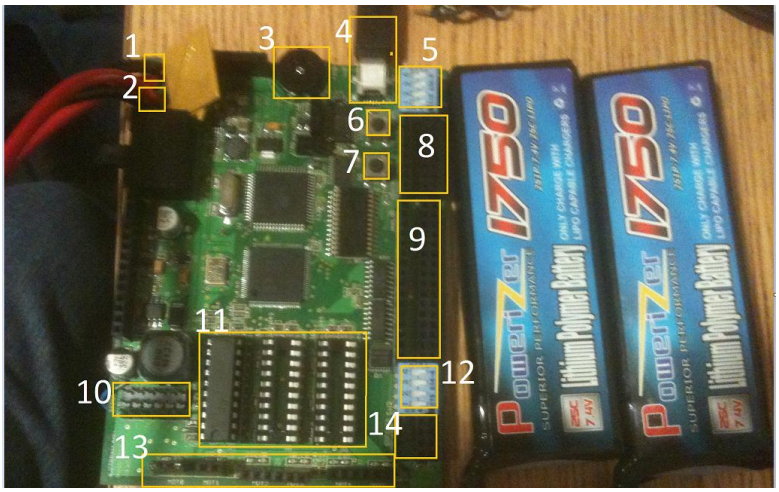


When plugging it to the motor ports, the motor doesn't really care which is positive and which is negative, it will operate regardless. However, it is highly recommended you select your own convention and stick to it. **Flipping the connection will flip the direction of the motor!!**

# 6.270 Happy Lab

**Programming a Motor:**

Look at your happyboard and locate your motor drivers. Look at box #11. The image below has only 1 Motor Driver on the left most side, so only motorpins 0 and 1 are activated.




Operating the motor only requires one command: **motor_set_vel(PORT_NUMBER, MOTOR_SPEED)**. Port Number is self explanatory which ranges from 0-5. MOTOR_SPEED ranges from -255 (full reverse) to 255(full forward) and 0 is stop.

Use **happytest** to see if your motor works.

**Open user/robot/umain.c** Try the following code:

```
// Entry point to contestant code.
int umain (void) {
    // YOUR CODE GOES HERE
    while (1) {
    motor_set_vel(0, 150);
    pause(1000);
    motor_set_vel(0, 255);
    pause(1000);
    motor_set_vel(0, 150);
    pause(1000);
    motor_set_vel(0, 0);
    pause(1000);
    motor_set_vel(0, -150);
    pause(1000);
    motor_set_vel(0, -255);
    pause(1000);
    motor_set_vel(0, -150);
    pause(1000);
    motor_set_vel(0, 0);
    pause(1000);
    }
    // Will never return, but the compiler complains without a return
    // statement.
    return 0;
}
```

Notice how the motor ramps up to 255 and ramps down to -255. It is highly advisable that you do a full stop before changing directions.

## 4.5 Standard Positional Servos

**Background Info:** Positional Servos are unique because as the name suggest, you can actively set the angular position of its output. The Servo has an internal **potentiometer** (a clever name for a variable resistor). There is a different resistance value for each positional output.

Although it is pretty neat to track the position of a servo output, there is one significant drawback. Servos can only turn from 0 to 180 degrees. Which means a positional servo cannot be geared without changing its internal structure. For this limitation, servos are incredibly strong but very slow. This follows the same rule as a DC Motor's Torque and RPM tradeoff. Servos are generally high torque and low RPM.

What the DC motor lacks in power and accuracy, the servo motor excels in these quality.

**Wiring:**

Fortunately, there is no need to wire anything. However, we should be careful what each wire is. Generally very dark wires indicate GND. Power is usually depicted as RED. For the Servos you're getting this year, the color scheme is: **BROWN (GND), RED(PWR) and Orange(Signal).**


Remember that Servos have a specific set of ports in the happy board. Refer to box #10 on **Intro to HappyBoard.** Make sure that you align GND, PWR, and signal when connecting the servo.

# 6.270 Happy Lab

**Programming**

Use happytest to try see if your servo works.

Alternatively, you can try writing your own snippet on **user/robot/umain.c**

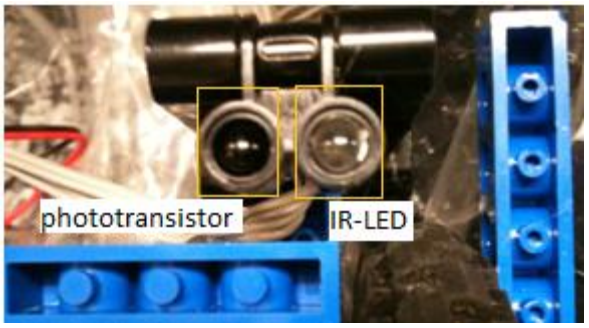
The command to control servos is: **servo_set_pos(ServoPort#, Position).** ServoPort ranges from 0 to 5. Position ranges from 0 to 511.

NOTE: Not all servos are alike. Be very careful about setting extreme positions near 0 or near 511 - this may be outside the motors position range, causing it to stall and eventually burn out!

```
// Entry point to contestant code.
int umain (void) {
   // YOUR CODE GOES HERE
    while (1) {
    servo_set_pos(0, 100);
    pause(1000);
    servo_set_pos(0, 256);
    pause(1000);
    servo_set_pos(0, 400);
    pause(1000);
    servo_set_pos(0, 256);
    pause(1000);
    }
    // Will never return, but the compiler complains without a return
    // statement.
    return 0;
}
```

## 4.6 IR LED and Phototransistor


LEFT: IR LED(2) + Phototransistor(1) Used as a wide break beam



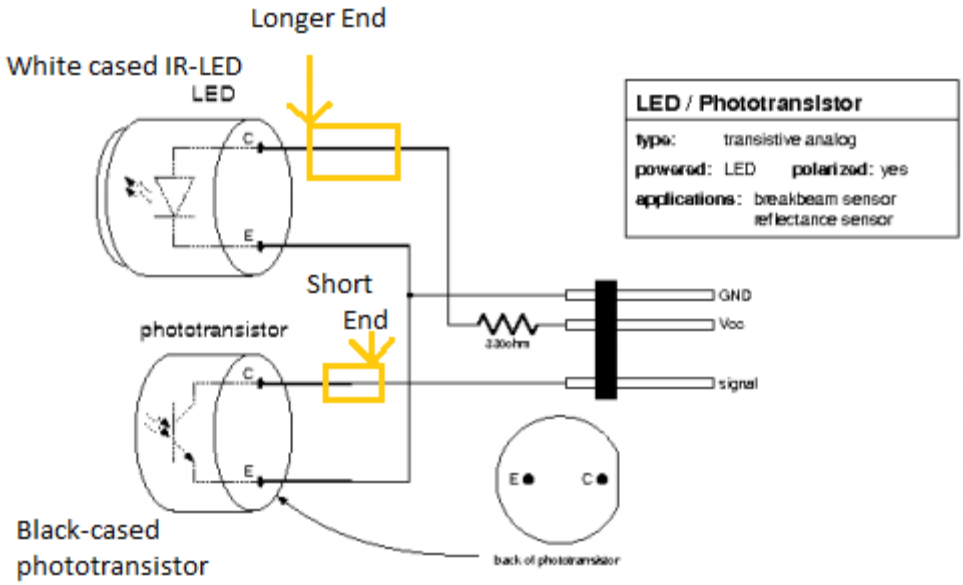

RIGHT: Used as a Line Follower/Light Follower

**Background Info:** The IR LED emits Infrared Light and the phototransistor receives those emissions either directly or through a reflection. The images above illustrate how these sensors are commonly used. The phototransistor is an **analog input sensor** so they can only be placed on pins 8-23.

The real sensor in the pair is the phototransistor which is sensitive to light. The more light that the phototransistor receivies, the lower its internal resistance will be. Similarly, the less light is available, the higher its internal resistance until no current can flow. The happyboard then translates the flow of current through varying internal resistances **as analog values.**

Although ambient lighting definitely affects the performance of the sensor and the analog values, the interaction between the IR-LED and phototransistor should give a significant difference to cancel out ambient lighting.

**Wiring:** If you haven't noticed it yet, you'll see that breakbeam encoders introduced much earlier are essentially a pair of IR LED and Phototransistors. Which means the wiring is going to be very similar.



# 6.270 Happy Lab

**Programming:** Reading analog values is called by the command analog_read(Port_Number)

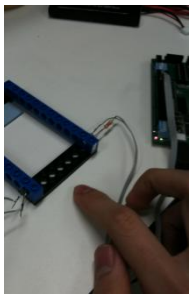
Use happytest to try see if your sensor works.

Try the following code.

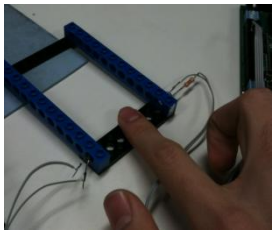
```
// Entry point to contestant code.
int umain (void) {
    // YOUR CODE GOES HERE
    while (1) {
        if (analog_read(15) > 500){
            printf("Analog Value: %d BEAM BREAKS!\n", analog_read(15));
        }
        else{
            printf("Analog Value: %d BEAM COMPLETES!\n", analog_read(15));
        }
    }
    // Will never return, but the compiler complains without a return
    // statement.
    return 0;
}
```

Note that 500 was an arbitrary value. You must figure out through testing how to best separate the values to determine whether the Infrared beam is broken beam or not.

Analog Value: 16 BEAM COMPLETES!
Analog Value: 16 BEAM COMPLETES!
Analog Value: 17 BEAM COMPLETES!



Analog Value: 883 BEAM BREAKS!
Analog Value: 876 BEAM BREAKS!
Analog Value: 891 BEAM BREAKS!
Analog Value: 892 BEAM BREAKS!



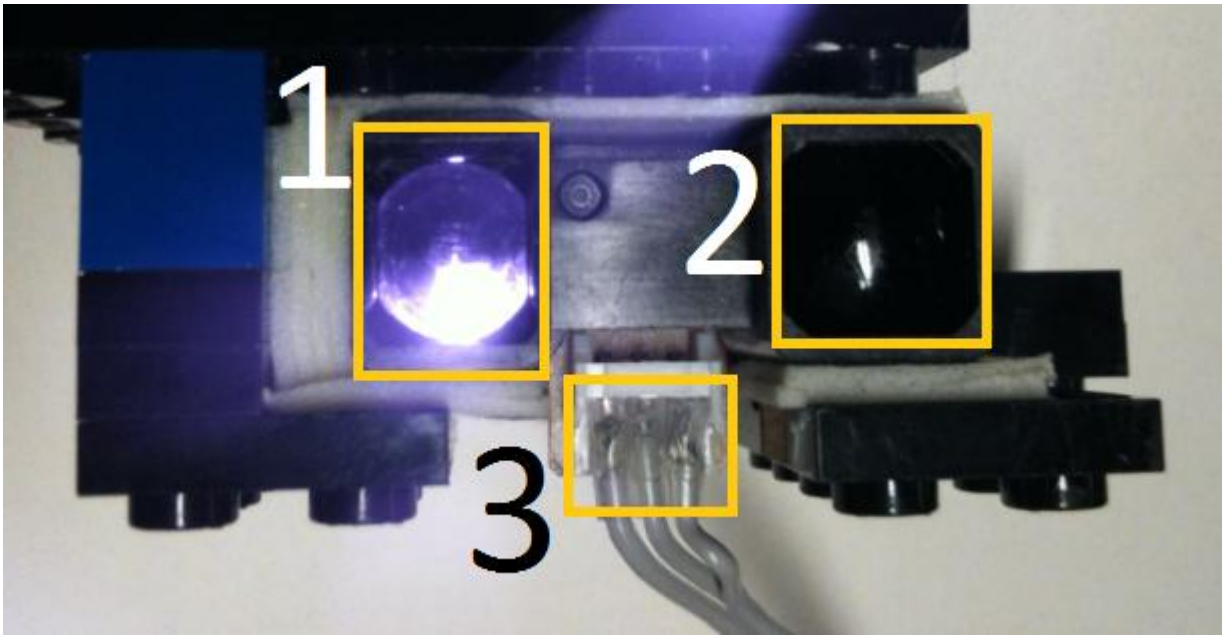
**When using the sensor as a break beam, use the holes in the Lego pieces to perfectly align the sensors.**

**Supplemental:** I hope you realize the power of the IR_LED and Phototransistor sensor. When it acts as a simple breakbeam, the analog values are easily seperable

But! You can use the varying analog values as an advantage. In this year's competition line following is pretty important. You can use an **array of IR LED & Phototransistors** to follow a line as well as use proportional control! Be creative and Goodluck!

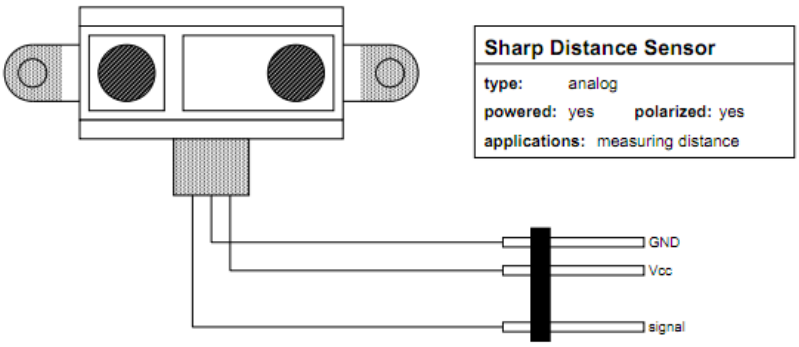
# 6.270 Happy Lab

## Supplemental 4.7 Sharp IR Distance Sensor




**Background Info:** A sharp IR distance sensor is a great way of measuring distances to walls or objects. A near infrared-light is emitted from box #1 and a detector on box#2 measures the angle of spot that the emitter projects on the wall. Sharp Distance sensors comes in different specifications The sharp distance sensor (http://www.sharpsma.com/webfm_send/1487 ) has a range from 8" to 60". It is important to note that this sensor **provides its own analog input.** Which means you can only put this sensor on **analog ports 20-23** and ensuring that the corresponding pull-up resistor (Box #12 on **Intro to HappyBoard)** is turned **OFF.**

**Wiring:** Looking at box #3, from left to right it is Signal, GND, VCC.




Remember that you can only put these sensors on ports **20-23** where you can turn off the pull-up resistor.

(**Alternatively**, you can use the other ports ONLY if you go into the PCB board and **manually remove the pullup-resistor trace**).

You can check and make sure that you soldered the correct wires by putting a camera right on top of the IR sensor to see that an infrared is actually being emitted.

**Programming:** Before the IR Sensor can be useful, we need to calibrate our IR Sensors. Plug anywhere from ports 20-23.

# 6.270 Happy Lab

**IR Distance sensor calibration:**

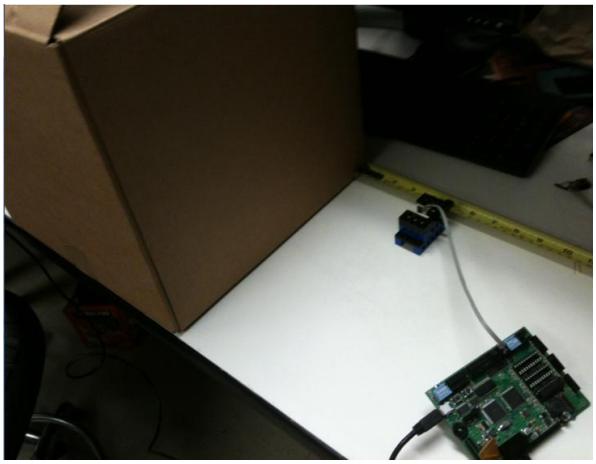
Navigate to **user/irdistcal/irdist.cal.c** Notice that there's an entire calibration algorithm written out.

Navigate back to the top of the joyos directory, open Makefile and change USERSRC to

**USERSRC = user/irdistcal/irdistcal.c**

Upload the program.

For calibration, have these extra materials with you: **Tape Measure, cardboard box, and a flat surface.**



For best readings, ensure that everything is perpendicular to each other.

Have happyboard connected to your computer. Open up your Serial terminal since there is a routine for calibration.

```
Running umain()...

IRDistCal      Press Go
Use frob to # ofsamples: 36
Use frob to # ofsamples: 36
Use frob to # ofsamples: 36
Use frob to # ofsamples: 36
Use frob to # ofsamples: 36
Use frob to # ofsamples: 36
```

1) **The Calibration will first ask how many samples you want to use for calibration. Use the frob knob to change the sample size. Try a size of *9*. (More samples is better though) Press GO on your happyboard when ready.**

```
Use frob to    select port: 20
Use frob to    select port: 20
Use frob to    select port: 20
```

2) **It then asks which port your sensor is plugged in. I used port 20. Remember that the pull-up resistor for 20 needs to be turned OFF. Use the frob know to select the port and press GO when ready**

```
Sample @ 10cm = 456
Sample @ 10cm = 456
Sample @ 10cm = 455
Sample @ 10cm = 457
Sample @ 10cm = 454
```

3) The code will now ask you to record a value @ 10cm away from the wall. Move the sensor to 10cm away from the wall and press GO when ready.

```
Sample @ 18cm = 532
Sample @ 18cm = 532
Sample @ 18cm = 531
```

4) It will keep asking you to do sample sizes at different distances. Keep giving the sensor samples until the program ends.

```
Sample @ 74cm = 161
Sample @ 74cm = 157
Sample @ 74cm = 171
Sample @ 74cm = 163
OK: M: 14421    C: 10, press Go
```

5) After the last sample you get to constants of M and C. In this case, I got M = 14421 and C = 10. Pressing GO again ends the calibration officially.

**IR Distance test code**

# 6.270 Happy Lab

Navigate to **user/iridisttest/iridisttest.c**

```
#include <joyos.h>
#include <happylib.h>

#define SHARP_M      22840
#define SHARP_C      26
#define SHARP_PORT   20

// usetup is called during the calibration period. It must return before the
// period ends.
int usetup (void) {
    irdist_set_calibration (SHARP_M, SHARP_C);
    return 0;
}

int umain(void) {
    for (;;) {
        printf ("dist = %.2f in\n", irdist_read (SHARP_PORT)/2.54);
        pause (40);
    }

    return 0;
}
```

1

2

3

4

There are a few key things to note about this code. Box#1 shows that we have a new library being imported called <happylib.h>. Box #2 has the constants which we found out through calibration. Box#3 shows the setup routine that is added right under usetup. Finally Box#4 shows how to get the sensor value itself. Notice that 2.54 is a conversion factor to INCHES. Irdist_read(PortNumber) actually returns distances in **CENTIMETERS**.

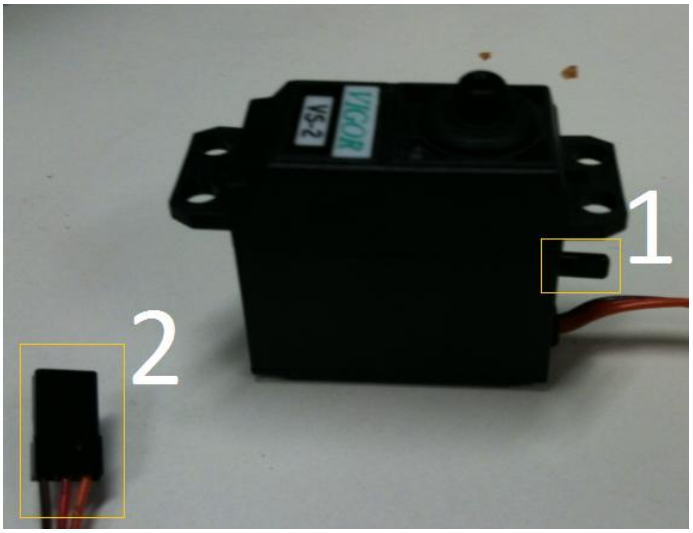
Upload this code by changing makefile: USERSRC = user/iridisttest/iridisttest.c

```
dist = 19.69 in
dist = 19.69 in
dist = 19.69 in
dist = 19.69 in
dist = 19.69 in
dist = 19.69 in
dist = 19.69 in
dist = 19.69 in
dist = 19.69 in
dist = 19.69 in
```

When I place mine at 19.5in, I get 19.69in. This is really good! However, when I move much closer, the error increases up to 3in. This is why it is important to give a greater sample size for the happyboard to calibrate.

# 6.270 Happy Lab

## 4.8 Supplemental: Continuous Servos




**Background Info:** There are many cases when It is preferable to use the low RPM and high torque of a **Positional Servo** without the 0-180 degree restriction. Opening up a traditional servo and removing the locking tab, frees up the restriction **but loses** its positional accuracy. In essence, it makes the high torque servo operate similar to a DC motor but with a ***serious catch.***

To make your servo continuous, please read Scott Bezek's tutotorial on **Servo Modification Guide.** Note that this is a **Permanent Change.**

After doing your modification, you will immediately notice that the servo's potentiometer is outside. This is marked as Box#1 on the image above. The potentiometer essentially sets the "center" of between turning Clockwise or Counterclockwise. Remember that the input to a servo ranges from 0-511. An off-the-shelf servo will have its potentiometer "center" at 256. However, once you take the servo apart, you will change this numerical center.

Why is it a problem? Suppose that the potentiometer is set all the way at one end. The new numerical center is at 511 (or 0 depending on which end). Which means, no matter what signal command you set from 0-511, the servo will always spin in the same direction.

It is important to calibrate to mechanically calibrate it so that it returns to a numerical center of 256.

# 6.270 Happy Lab

**Wiring**

Box#2 on the image above shows that the continuous servo wires the same as a traditional positional servo.

**Calibrating a Continuous Servo**

Upload the following program:

```
// Entry point to contestant code.
int umain (void) {
   // YOUR CODE GOES HERE
    while (1) {
    servo_set_pos(0, 256);
    }
    // Will never return, but the compiler complains without a return
    // statement.
    return 0;
}
```

Move around the potentiometer until the servo completely halts. It is at a very precise location so be careful. Notice that overshooting in either direction causes the center to change which in turn rotates the servo output.

When you finally found the center of the potentiometer, grab a hot glue and put this in place.

**Programming a Servo**

The programming of a continuous servo is essentially the same as a positional servo. After calibration, your servo now acts exactly as a DC motor with *pseudo-PWM qualities*. “pseudo” because the servo itself has a high proportional gain which makes noticeable speed changes only near the center value of 256. You should be very careful setting values near 256 because the potentiometer center could potentially drift with vibration.

**servo_set_pos(#Port, #Value)**

Values from 0-255 spin the servo in the negative direction with 0 as the greatest value.

Values from 257-511 spins the servo in the positive direction with 511 as the greatest value.

**servo_disable(#port)**

This year, servo_disabled() is a new command which completely halts the motion of the continuous servo no matter where the numerical center is.

# 6.270 Happy Lab

## Step 5.0 Final TIPS

The purpose of happy lab is to get you going as fast as possible by walking you step by step. BUT! There is still plenty of material left to cover.

Here are some final tips:

1. **Go to Lectures**! There's only 8 of them over the span of 5 days. You will get everything you need to know in terms of background knowledge in these lectures.

2. **READ!** There's a lot of material on the 6.270 website about almost everything need to succeed in the class and competition. The **2009 6.270 course notes** are still applicable. Make sure you read through the mechanical sections of the notes. You can also find more about the **JoyOS library** on the main 6.270 websites. For example, we didn't cover how to do **digital_write(Port#)** but all of these information are available in the library.

3. **Go to required LABS! –** You get an instant jump start and organizers are there to make sure you never make the mistake they did.

4. **Go to LAB anyway!** Typically, the people who succeed and get a lot out of 6.270 are those who put in the extra time to be in lab. Actually, all past winners have been in lab at least 10 hours a day, every day.

5. **ASK Questions!** Are you stumped? Ask questions ASAP. There's no need to brood over a problem when an organizer who knows the answer is only 10ft away.

   **Luke O'Malley's Tips:**

6. You must be able to **step back and take a deep breath**. Throughout the month take some time to explore Boston, see a movie, go to the museum, etc. It is easy to lose site of your project when it is what you are living and breathing everyday. Some of my best ideas for our robot came from getting out and changing pace.

7. Being frantic and **rushing a decision is dangerous**. You are all engineers. Know why you are doing what is you are doing and be able to explain your decision. This is particularly relevant to trying to find bugs in code.

8. **Use version control.** The integrity of large scale software projects cannot be guaranteed without it. It will provide peace of mind and save you from potentially devastating code bugs. My team had working bits of code that kept getting modified throughout the month until we had no working code. When using git, take advantage of the notes. Make sure to mark milestones.

9. **Don't put everything in one master file.** Segment your code.

10. **Use meaningful variable names and comment well**. 30 days from now you want to know why you put in that weird constant that doesn't seem to do anything.

11. **Have a game plan**. You must make sure your team works in concert. All of the parts have to move together. Give your programmer a basic chassis to test code on. There is no reason why coding should hold up construction and vice versa. This happened to my team and was lethal. Go for milestones. Build upon success. Get a robot that can

navigate to given coordinates. Then get a robot that can avoid collision. Tie these functions together. Draw, talk, and flush out ideas with your team. A lot of MIT students are very good at doing everything inside their heads and working alone. You are a single cog in a bigger machine for the month.

12. **If you are going to use threading, test thoroughly for race conditions and resource competition.** You will never be able to predicate all of the behaviors. Threading is great for utilizing down time (ie. While driving to a destination, you could be doing collision detection and objective analysis).

13. **Test often**. A specific test suite/procedure for validation of your code and mechanics could help identify bugs and establish a means for comparing different iterations of your robot.

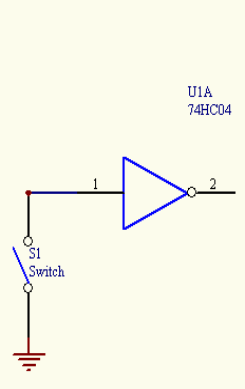
14. **Abstract.** Functions are your friend. You want to put in the time now for the foundation so you can worry about the high level stuff as the final competition approaches.

# 6.270 Happy Lab

## Appendix A: Pull-Up Resistors

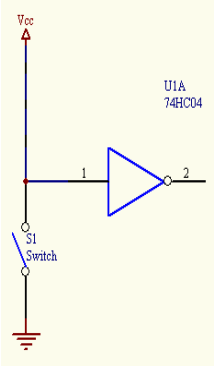
The Following Images are from: http://www.seattlerobotics.org/encoder/mar97/basics.html

Logic Gates are highly susceptible to electrical noise. For example, the Image below does not have a pull-up resistor So its logic value is **floating** and can alternate between (1 or 0).



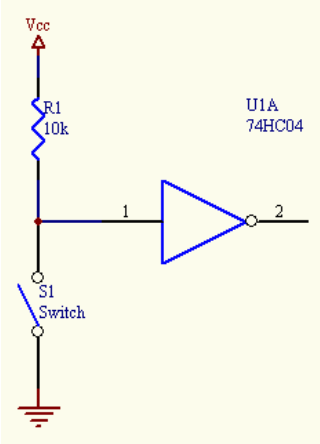

BAD: Floating Logic Gate
Susceptible to Electrical Noise

The Configuration Below partially solves the problem because Vcc always provides a HIGH value., but when the switch is closed, it causes a short.



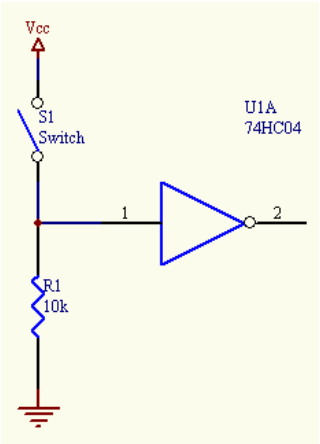

BAD: Solves Problem until switch is closed leading to a short

Now with a pull-up/pull down resistor, the logic gates always have a value on both cases when the switch is ON and OFF.

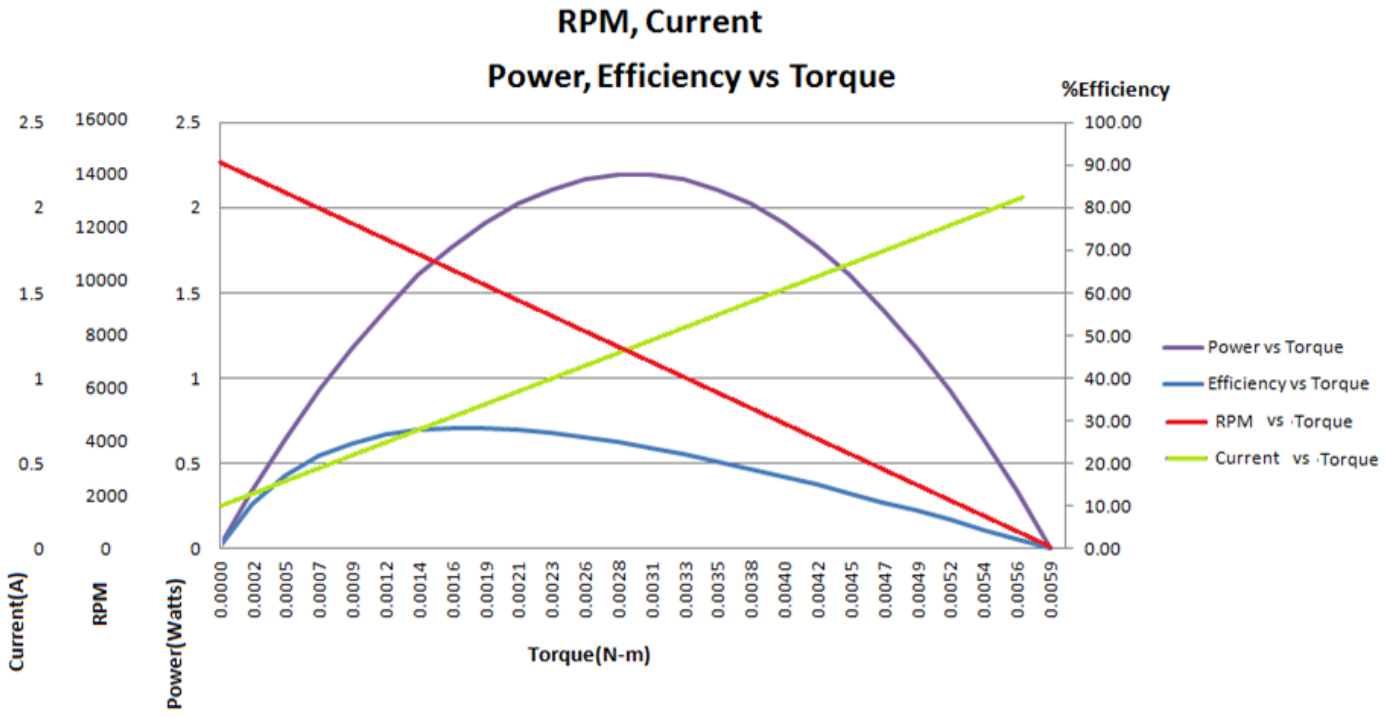



Left: Pull Up Resistor; When Switch is off, Logic receives HIGH input

Right: Pull Down Resistor; When Switch is off, Logic receives LOW as input

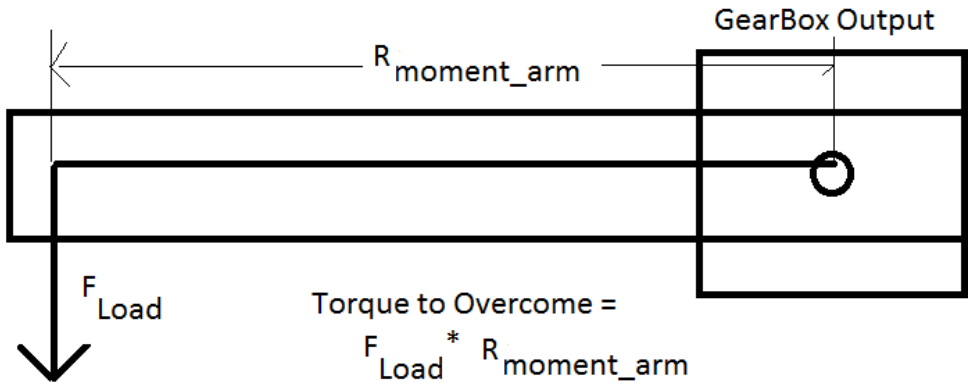


# Appendix B: Using the Motor Curve as a Design Tool




Suppose we have one of the linkages of a dumping mechanism. The motor needs to overcome a torque of

$T = F_{Load} * R_{moment\_arm}$ . Let us now assume this torque is **0.25Nm.**




If we were to plot this torque on our motor curve above (See Graph), we can definitely see that 0.5N-m is out of the scale. The motor cannot rotate against this torque at all.

This is where Gear Ratios come into play. **Gear Ratios can scale and reflect a lower torque on the motor.** But how exactly do we scale our Gear Ratios given a load torque?

# 6.270 Happy Lab

First we must understand what these values mean

| | Values | |
|---|---|---|
| **Free Speed** | 14292 | RPM |
| **Free Current** | 0.39 | Amps |
| **Stall Current** | 2.2 | Amps |
| **Stall Torque** | 0.00587 | N-m |

Operating Voltage: 7V

**Free Speed:** RPM of motor at 0 torque.
**Free Current:** Current the motor draws at 0 Torque
**Stall Current:** Current the motor draws at stall torque.
**Stall Torque:** Torque at which the motor has 0 RPM. Motor does 0 work.

Looking at the Motor Curve above, a **reflected torque of ~25%** of the stall gives maximum motor efficiency. So We need to find a value for the Gear Ratio such that the reflected torque on the motor is
**Required Reflected Torque = 0.25*Stall Torque = 0.0014675 N-m**

$$\frac{N_{out}}{N_{in}} = \frac{W_{in}}{W_{out}} = \frac{T_{out}}{T_{in}}$$

Remember that

So Our Gear Ratio should be: $\frac{Nout}{Nin} = \frac{T_{out}}{T_{in}} = \frac{0.25Nm}{0.001475Nm} = 170{:}\,1\ Gear\ Reduction$

With this gear Ratio, Our Torque output is now 170 times greater than before, but out RPM output is 170 times less.

Torque output at End of Gearbox = 0.251Nm RPM Output at End of gearbox =84RPM

But the Tradeoff is well worth it because we can carry a heavy load at a decent RPM.

<u>HOW PWM Affects Motor Performance:</u>

**PWM (Pulse-Width Modulation)** – Fast ON and OFF switching of the motor. Depending on how long the motor is turned ON during each cycle, the average voltage sent to the motor can be anywhere from 0%-100% of the original. A 75% duty cycle means that the motor is Turned ON 75% of the time (average sent voltage is 75%). In essence, PWM scales the voltage sent to the motor.

*Using PWM scales your operating voltage down*. If you operate at 50% voltage (50% duty cycle) your total output torque and speed goes down by 50%. **You do not get extra torque with a lower PWM!** When you are calculating gear ratios, take note at which PWM *duty cycle* you are operating the motor.

**This is why engineers usually scale the gear ratio even further from the exact calculated value since we want to have an extra safety and confidence factor. Remember also that the motor battery also decreases during operation.**

Further Reading: Understanding the DC Motor: http://lancet.mit.edu/motors/motors3.html