EE40 Final Project-1 Smart Car

Name & SID 1 :

Name & SID 2:

Introduction

The final project is to create an intelligent vehicle, better known as a robot. You will be provided with a chassis(motorized base), which has breadboard space for you to build circuits, and a microcontroller. As in lab 6, you will be writing the program that tells the robot what to do, but you will also build the interface circuits that enable the microcontroller to control the robot's actions. Like an actual engineering job, you will plan the design of your circuits and gain the approval of your boss (your GSI) before actually building your circuits.

Power Source

Because a robot must move around and availability of short cables, we can't use the plug power that we usually do in the lab. This means we need to use batteries to power everything on the robot. Since motors consume a lot more power than the microcontroller, using separate batteries for the motors and the microcontroller lets us use bigger batteries and higher voltages for the motors without damaging the microcontroller.

Be extra careful to never short out the motor batteries. They have no "output off" setting and no current limit, and will happily try to melt any wires, breadboard rows, ICs, or tools you short their terminals with! Measure the resistance of the motor. Now based on this resistance value along with the voltage that the battery can supply and the energy(mA.hour) that the battery provides, find out how long would you expect the motor batteries to be able to continuously power the two motors if they don't spin?

(1pt)

Motor Resistance	
Battery Voltage	
Battery Energy	
Stand-by Time	

Motor Driver

if we run the motors from a separate power source, we can't connect the microcontroller directly to the motors, so an interface is required. We will be using a classic motor interface design called an H-bridge, which can be bought as a single IC for convenience. It lets us run a motor in either direction easily.

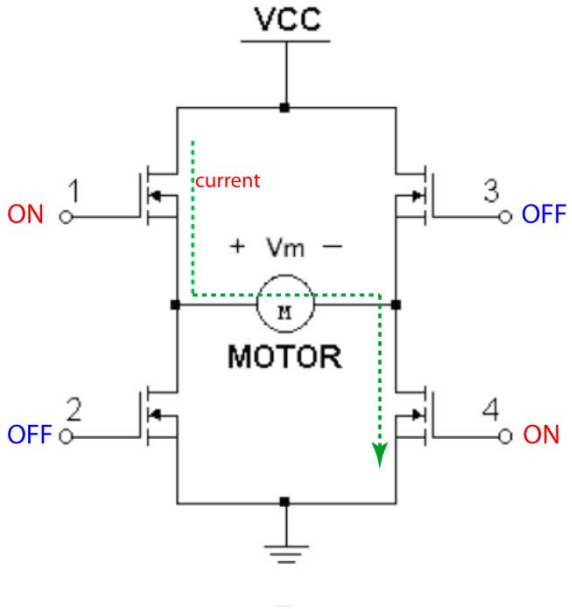
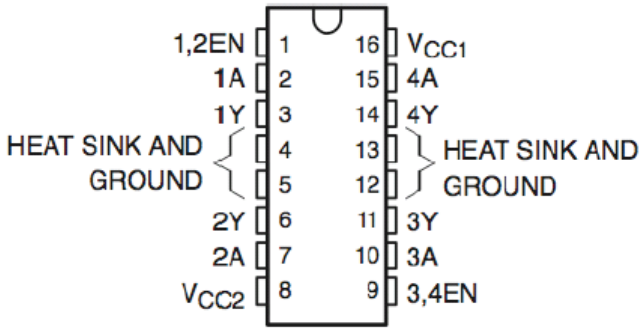


Figure 1. A basic Hbridge IC using MOSFETs

DC motors like the ones we use have a sort of polarization. If the polarity of the applied voltage is flipped, they will rotate in the opposite direction. An H-bridge is a convenient way of producing this polarity flip. By turning on one transistor on each side, we can make the motor voltage *Vm* positive or negative. Can you see a problem with this simple design that can arise if we aren't careful with inputs?

(1pt)

We can avoid this problem by using a purpose-built motor driver chip that uses the same general design but prevents problematic inputs from being possible in the first place. We are using the SN754410, which contains four "sides" of an Hbridge on a single chip. Each input on the chip controls whether the corresponding output is connected to Vcc or ground. The major problem we have with the SN754410 is that it cannot take its inputs directly from the microcontroller. If you look at its data sheet, you will see that its minimum allowed input voltage is larger than the maximum voltage that the microcontroller can put out.



FUNCTION TABLE
(each driver)

INPUTS†		OUTPUT
A	EN	Y
H	H	H
L	H	L
X	L	Z

H = high-level, L = low-level
X = irrelevant
Z = high-impedance (off)

Figure 2. Pinout and function of the SN754410

This problem can be solved by "translating" the 3V signals produced by the microcontroller into higher-voltage signals. The motor batteries provide 4.8V, which is a sufficiently high voltage for this purpose. There are a variety of circuits capable of achieving this; in this lab, we will use op-amps. We could build an amplifier, but that is actually more complicated than we need. Remember that an op-amp magnifies the difference between its inputs until its output reaches its supply voltage. Can you use this property to design a simple circuit to convert a 0-3V digital signal into a 0-4.8V digital signal, without using any feedback? Be sure to include some tolerance for inputs that aren't exactly zero. This circuit is called a comparator. **(2pt)**

Now, put it all together. Draw a schematic of one entire motor driving circuit, starting with two microcontroller pins (0 to 3 volt range, remember) and ending with a motor. This will form half of your robot's motion system. Be sure to connect every pin on the half of the SN754410 you're using. You'll be using LMC6482 dual op-amp chips for your operational amplifier needs. **(1pt)**

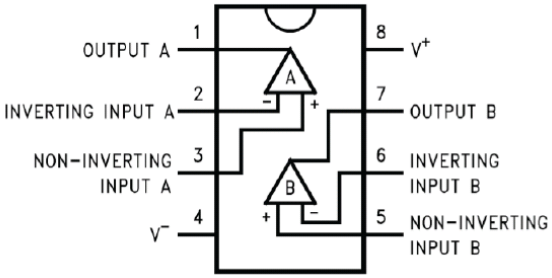


Figure 3. Pinout of the LMC6482

Get approval of your circuit from your GSI, then get the parts and build the circuit on your robot's breadboard.

Control Software

Having the hardware we need is nice. Now for the software. To start off, let's have the robot only move forward, move backward, or stand still. Port 10's pins are in a convenient place on the microcontroller board, so we'll use them to drive the motors. To do this, we need to both set up the pins we need as outputs, and create functions to move the robot around. The following code configures Port 10, pins 0 through 3 as outputs:

```
P10OUT &= ~(BIT0+BIT1+BIT2+BIT3); // P10.0-3 low
P10SEL = 0x00; // All of port 10 set up for digital I/O
P10DIR |= BIT0+BIT1+BIT2+BIT3; // configure P10.0 as output
```

Now create functions you can use to move the robot around. All these functions need to do is move the motors in the right directions; your `main()` function can decide what to do after that. Don't forget, the motors might already be moving!

```
void forward()              (1pt)
{
```

```
}
void backward()             (1pt)
{
```

```
}
void left()                 (1pt)
{
```

```
}
void right()                (1pt)
{
```

```
}
```

A `stop()` function would probably also be useful.

Now that you can program your robot to move around, have it do something neat: move back and forth, spin around, whatever you like. You may find it useful to add the old `delay()` function to your code and use it to control the length of time that the robot moves in a given direction:

```
void delay(unsigned int n)
{
while (n > 0) n--;
}
```

Once you've put together a routine for your robot, show it to the GSI.

Sensing

We now have a fairly basic motorized vehicle. However, this is not much of a robot - it has no ability to observe the world and react. We will add this ability with the pair of switches on the front.
First we need to add the code that enables the microcontroller to detect the state of the switches. If they are connected to pins 6 and 7 of Port 10, that code is:

```
P10OUT |= BIT6+BIT7; // set P10.6-7 high
P10DIR &= ~(BIT6+BIT7); // P10.6-7 set as outputs
P10REN |= BIT6+BIT7; // activate pull-up resistors for
P10.6-7
```

Now the state of the switch on pin 6 (for example) can be used to make decisions. We can choose between two options as shown below:

```
if (~P10IN & BIT6) // is P10.6 low? (is the switch
pressed?)
{
// switch is pressed, do the stuff in these curly braces
}
else // optional
{
// switch is released, do the stuff in these braces instead
}
```

A fairly simple yet useful way of using this information is obstacle avoidance. In other words, if the robot runs into something, back up and try to avoid it. This will allow the robot to run around freely and not get stuck (assuming it doesn't fall off of any ledges).
Consider: If the robot hits something with its left switch, which way should it go after it backs up, in order to avoid the obstacle?

Write your main loop here. **(2pt)**

Soldering the Switches

Now that the software is ready, we need to connect the actual switches to the microcontroller. It's possible to do that with connectors that slide over the tabs on the switches, but we don't have any of those, so it's time for plan B: solder. You have seen soldered connections before; every circuit board has them. Solder is a mixture of metals that has a relatively low melting point, letting us use it to electrically connect two conductors. (It is not, however, conductive glue; you must have a sturdy physical connection before adding solder. This will also keep the two things you're soldering from slipping, which weakens the solder's bond.) Using a soldering iron is fairly simple. Turn it on and wait a few minutes for it to heat up. Once it's hot, put a bit of solder on the tip to improve heat transfer. To make a connection, place it where the side of the tip touches both things you want to solder. (The very tip doesn't have enough surface area for good heat transfer, so you need to use the side of the tip.) Place your solder in another place where it touches both things you want to solder, but not the soldering iron itself; we only want the solder to melt when the things we are soldering together are hot enough for the solder to make a good bond to them. After enough solder to make a joint is applied (the surface of the solder should be concave, not convex; a blob is too much), take the solder and iron away from the joint. Don't blow on the joint to cool it faster; it causes uneven cooling and a poor connection.

Naturally, if you're using a heated bit of metal to melt another bit of metal, they're both pretty hot and you don't want to touch them. A few hundred degrees is much hotter than you want touching your skin. Even if the soldering iron has been off for a while, hold your hand just above it to make sure it's actually cool before seizing the metal end for whatever reason. Finally, please don't solder anything on the microcontroller board, as its components are a lot more sensitive (and a lot smaller) than a switch and a wire. When soldering the wires to the switches, be sure to connect wires to the right terminals. On one side of each switch is a set of abbreviations showing what each connection point does when the switch is pressed; you'll need to connect to COM (common) and NO ("normally open", connects to COM when switch is pressed), but you don't really need NC ("normally closed", opens normal connection to COM when switch is pressed). Also, make sure your wires are long enough to get to the breadboard.

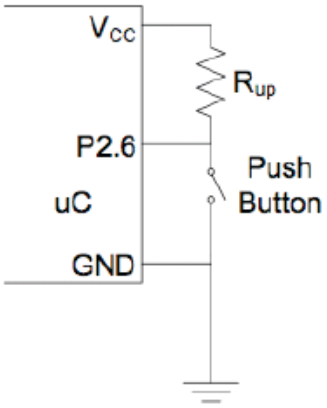


Figure 4. A sample microcontroller input with enabled pullup

Connecting the Switches

These input pins are just like the one connected to the button on the microcontroller board in that they have pull-up resistors built in. (You may recall adding a line of code to enable them. Go look!) This makes connecting the switches to the microcontroller extremely simple, as shown in the figure. Hook them up to match the choices for left and right you made while programming.

Once your robot can successfully hit and avoid an obstacle by itself, have it verified by the GSI.

Gradual Motion

You may have noticed that your robot moves very jerkily. Whenever the motors are turned on or off, there is an abrupt change in velocity. We can make it start and stop more gradually, or move more slowly if need be, by using a technique called PWM (pulse width modulation). It's often used as a simple digital-to analog

converter, and is also used in the control of lights and motors. We now apply the concept to the robot. You may remember the following code:

```
void dac(unsigned int n) {
... set output high ...
delay(n);
... set output low ...
delay(256-n);            }
```

The above code was for a single output. For the robot to accelerate and decelerate smoothly, you will need to write four functions: `accel-forward`, `decel-forward`, `accel-backward`, and `decel-backward`. (Turning is mostly around the center of mass, and we won't worry about making it smooth.) Each function should smoothly increase or decrease the speed of the motors in the appropriate direction; `accel-forward` should smoothly increase from zero to maximum forward speed, `decel-forward` should smoothly decrease from maximum forward speed to zero, and so on. The acceleration and deceleration should occur over about a second each. A helpful type of statement for this task is the `for` loop. It will run a command or group of commands over and over, changing a specific variable's value each time. For example, look at the following code:

```
for(unsigned int x = 60000; x >= 10000; x = x-500)
{
P1OUT ^= BIT0;
delay(x);
}
for(int i = 0; i < 10; i = i+1)
{
delay(500);
}
```

The upper loop will blink the LED on P1.0, starting with a slow blink and increasing to 6 times the starting speed before stopping when the value of `x` goes below 10000. The lower loop is effectively `delay(5000)`; every time around the loop, the value of `i` is increased by one until it is equal to 10, at which point the loop ends. (This is useful for creating delays much longer than the maximum delay `delay(65535)`, if you need one. Also, without the `unsigned` before the `int`, the limit would be 32767.) Basically, a `for` loop works like this:

```
for([do this once at the beginning];
[this must be true for the loop to repeat];
[do this each time the loop repeats])
{
}
```

Write your code for `accel-forward(unsigned int n)` below: **(2pt)**

Set your `main()` function to make the robot gradually accelerate and decelerate as it moves forward, then backward. If you're feeling artistic, add some turns. Show it to your GSI.

Motor Voltage Booster

The motor batteries will operate the motors, but the motors run fairly slowly. They would be faster if we increase the motor voltage, but adding batteries means adding weight, which would slow the robot down. We can increase the voltage without adding much weight by boosting the voltage with a boost converter. Remember, though, that we can't get power from nowhere - if we increase the voltage to the motors, we will drain more current from the battery.

If we add a boost converter that doubles the motor voltage but adds a 10% current overhead, how long will the battery last if the two motors never spin? Is this likely to be a problem for usability? **(1pt)**

To add a boost converter to a mobile platform, we can't use the function generator, so we'll have to use a 555 oscillator. Using what you learned in previous labs about the 555 timer and boost converters, design a boost converter that is self-contained. Show the design to your GSI before building anything. The new motor voltage should be connected to the Vcc2 (output Vcc) pin of the SN754410, where it will be distributed to the motors, and *not* the Vcc1 (input Vcc) pin, where it may damage the op-amps or the SN754410 itself. Put your design on the back of this page for four (4) points (if it works, verified by GSI, of course).