Jeff Page

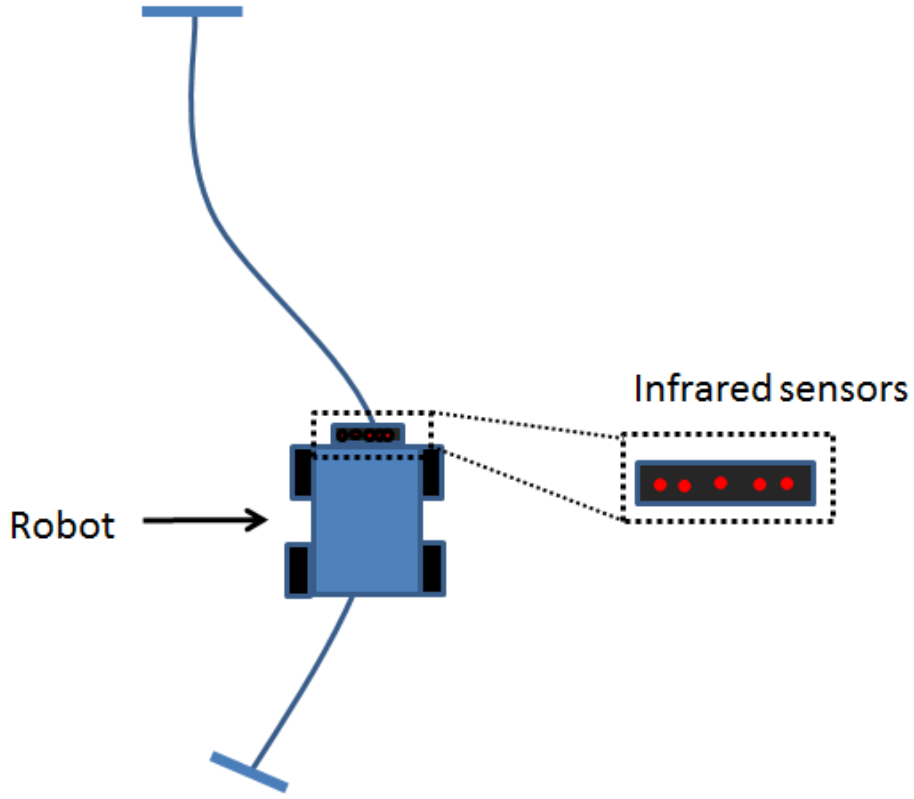
Oakland University Fall 2014

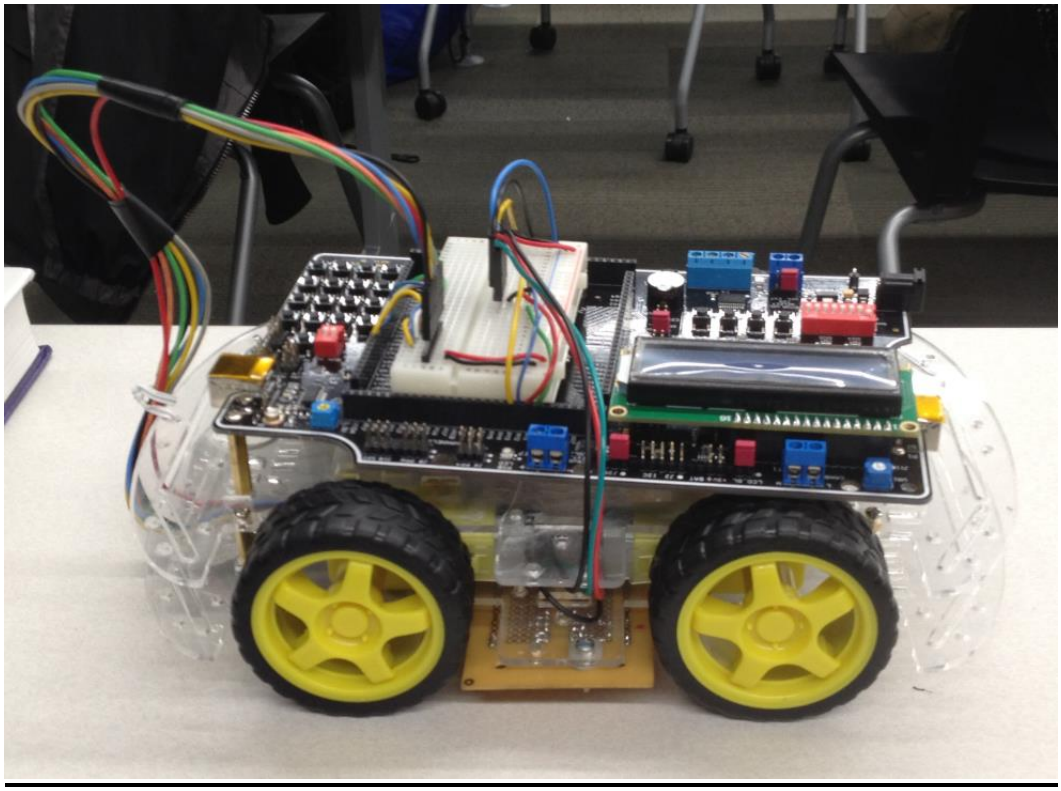
Project for ECE 570

PWM motor controlled line following robot

Introduction

The main object is to design a robot that can follow a reflective line that is placed on a carpeted surface. The robot has a chassis, 4 electric motors, an H bridge, the dragon12-Light board and a handmade infrared light sensor(with 5 outputs).





The Line Sensor

The sensor originally consisted of; Printed Circuit Board, five infrared LEDs, five photo transistors. It also had ten resistors, five 120 Ohms and five 11k Ohm resistors. The image blow shows the diagram of the circuit.

R1 = 120 Ohms

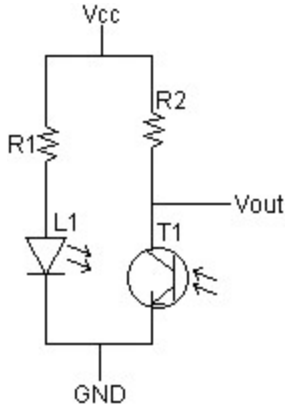
R2 = 11k Ohms

L1 – Infrared LED

T1 – Photo Transistor

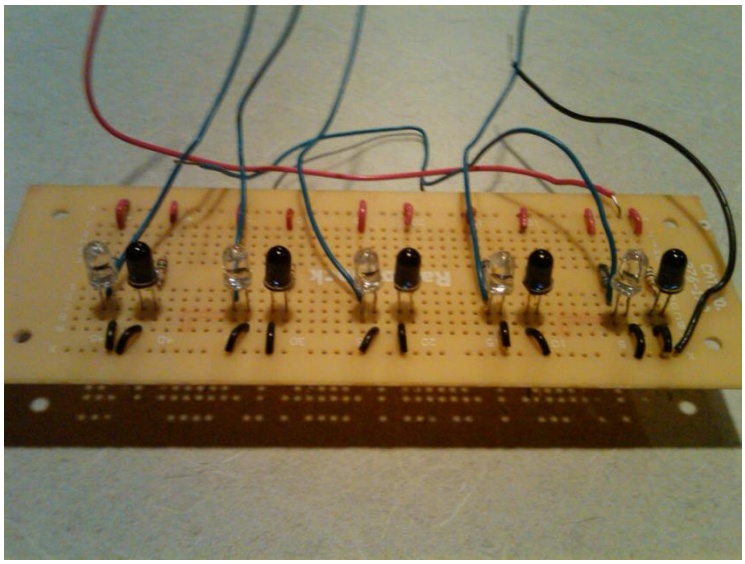
Vcc – 5 Volt Input

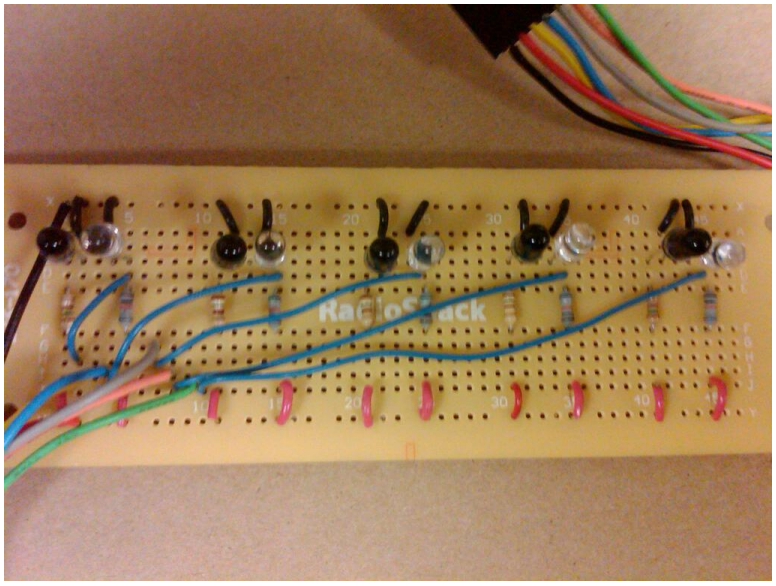
GND – Ground



Vout – Voltage Output

The voltage output ranged from 4.8 to 4.95 volts depending on whether the sensor was on white tape or on the carpet. This voltage output was not ideal for the A to D converter because there is not a large difference in the voltages. Due to this I created an Op amp to provide a 0-5V output. The actual output ended up being about 0.5-5V. This was a much more ideal than the voltage output that previously provided by the sensor. The Op Amp circuit that I designed is shown below.





R1 = 150 Ohms

R2 = 220k Ohms

R3 = 4.7k Ohms

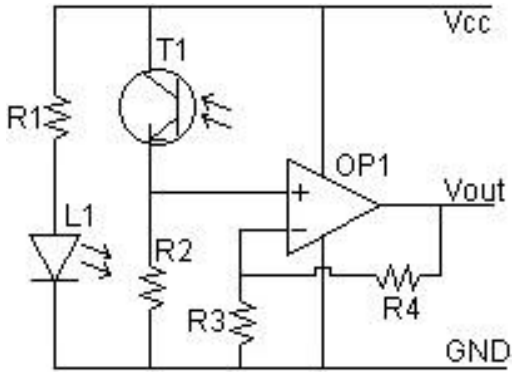
R4 = 10k Ohms

OP1 – LM 358 Operating Amplifier

Vcc – 5 Volt input

GND – Ground

Vout - Voltage Output



1 x infrared emitter - 2V 40mA

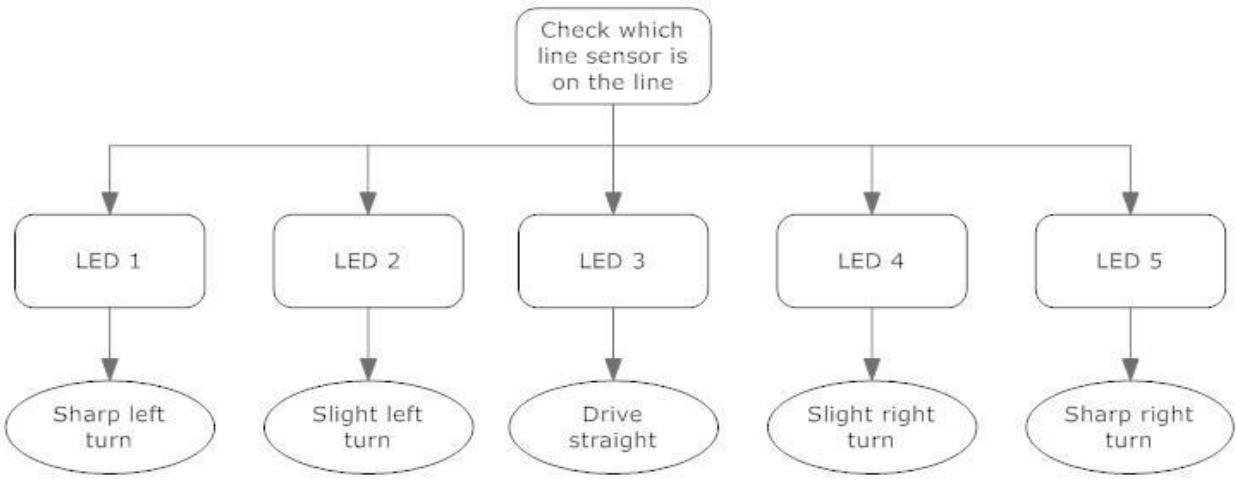
1 x phototransistor detector - 20V 25mA

Creating a comparator in C language I can tell which sensor is above the reflective tape.

```
led[0] = ATD1DR0;
led[1] = ATD1DR1;
led[2] = ATD1DR2;
led[3] = ATD1DR3;
led[4] = ATD1DR4;
/PORTB = 0x01;

if(led[0] > led[1] && led[0] > led[4] && led[0] > led[2] && led[0] > led[3]){//&&
}else if(led[1] > led[2] && led[1] > led[3] && led[1] > led[4] && led[3] > led[0])
}else if(led[2] > led[1] && led[2] > led[3] && led[2] > led[4] && led[2] > led[0])
}else if(led[3] > led[1] && led[3] > led[2] && led[3] > led[4] && led[3] > led[0])
}else if(led[4] > led[1] && led[4] > led[3] && led[4] > led[2] && led[4] > led[0])
}
```

A schematic is shown below for deciding which motor should be driven during the operation.



Experimental Setup

To test smaller parts of my code I used tools to make sure that the code was working. To ensure that the sensor was working I used a multi-meter test the voltage output. To test that my code was working for the A to D converters and the comparator I lit the LEDs on the Dragonboard to say which one is active. To test the PWM output to the H bridge I used a mult-meter to make sure that I was getting the voltage level that I wanted.

Results

The result of the project was that I created a robot that could tell where the line was relative to the right or left side of the car. This was shown on the LEDs of the board. Then I sent a PWM output to the H bridge to drive the motors. I had some issues with the voltage levels and fine

tuning of the H bridge to get the motors to work the way that I wanted them to but they do show the correct direction for which the robot should travel.

Conclusion

The key take away and improvements that could be made are with respect to the H bridge control that I previously mentioned. The H bridge provided motor control that was either too high or too low based on the voltage source that I applied to it. I think that a bit more time with the dine tuning of the PWM source in the code and finding the proper voltage source that the robot would have worked very nicely.

Sources

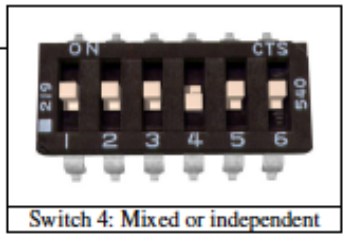
The H Bridge Spec Sheet

Mode 1: Analog Input

Analog input mode is selected by setting switches 1 and 2 to the UP position. Switch 3 should be either up or down, depending on the battery type being used. Inputs S1 and S2 are configured as analog inputs. The output impedance of the signals fed into the inputs should be less than 10k ohms for best results. If you are using a potentiometer to generate the input signals, a 1k, 5k or 10k linear taper pot is recommended. In all cases, an analog voltage of 2.5V corresponds to no movement. Signals above 2.5V will command a forward motion and signals below 2.5V will command a backwards motion.

There are three operating options for analog input. These are selected with switches 4, 5 and 6. All the options can be used independently or in any combination.

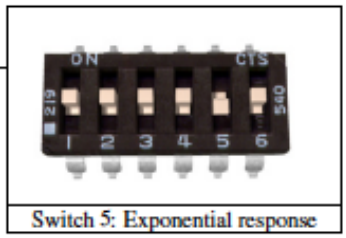
Switch 4: Mixing Mode



Switch 4: Mixed or independent

If switch 4 is in the UP position, the Sabertooth 2x10 is in **Mixed** mode. This mode is designed for easy steering of differential-drive vehicles. The analog signal fed into S1 controls the forward/back motion of the vehicle, and the analog signal fed into S2 controls the turning motion of the vehicle. If Switch 4 is in the DOWN position, the Sabertooth 2x10 is in Independent mode. In Independent mode, the signal fed to S1 directly controls Motor 1 (outputs M1A and M1B) and the signal fed to S2 controls Motor 2.

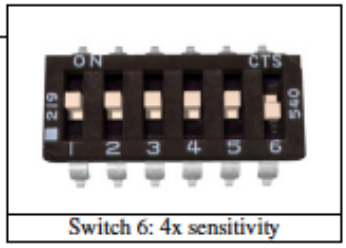
Switch 5: Exponential response



Switch 5: Exponential response

If switch 5 is in the DOWN position, the response to input signals will be exponential. This softens control around the zero speed point, which is useful for control of vehicles with fast top speeds or fast max turning rates. If switch 5 is in the UP position, the response is linear.

Switch 6: 4x sensitivity



Switch 6: 4x sensitivity

If switch 6 is in the UP position, the input signal range is from 0v to 5v, with a zero point of 2.5v.
If switch 6 is in the DOWN position, 4x sensitivity mode is enabled. In this mode, the input signal range is from 1.875V to 3.125V, with a zero point of 2.5v. This is useful for building analog feedback loops

Note on using filtered PWM in Analog Mode

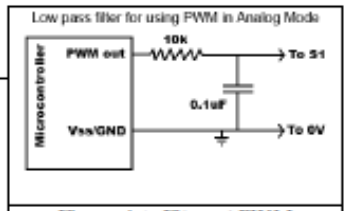


Figure 4.1: Filtered PWM

If you are using a filtered PWM signal from a microcontroller to generate the analog voltage, an R/C filter with component values 10k ohms and at least .1uf is recommended as shown in **Figure 4.1.** Using a larger value filter capacitor such as 1uf or 10uf will result in smoother motor operation, at a cost of slower transient response. A PWM frequency higher than 1000Hz is recommended.

Mode 2: R/C Input

R/C input mode is used with a standard hobby Radio control transmitter and receiver, or a microcontroller using the same protocol. R/C mode is selected by setting switch 1 to the DOWN position and switch 2 to the UP position. If running from a receiver, it is necessary to obtain one or more servo pigtails and hook them up according to figure 5.1. If there are only motor drivers being used it is acceptable to power the receiver or microcontroller directly from the Sabertooth as shown. If the system also has to power servos or other 5v loads, we recommend a ParkBEC or a receiver battery pack, as shown in figure 5.2. If using a receiver pack, do not connect power to the 5V line of the Sabertooth because the maximum voltage it can tolerate is 6V.

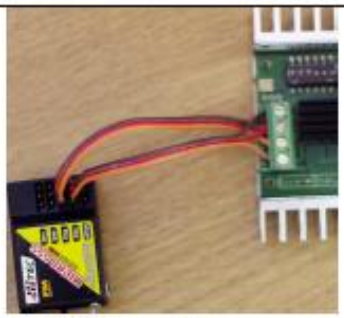


Figure 5.1: R/C connection

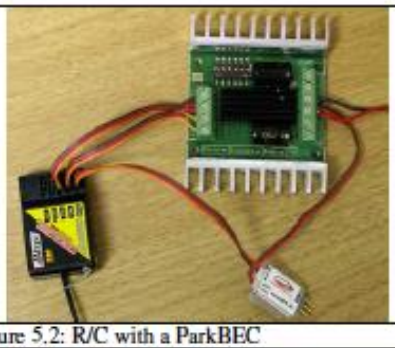


Figure 5.2: R/C with a ParkBEC

There are three operating options for R/C mode. These are selected with switches 4, 5 and 6.

Switch 4: Mixing Mode

R/C Mixed or Independent

When Switch 4 is in the UP position, Mixed mode is selected. In this mode, the R/C signal fed to the S1 input controls the forward/backwards motion of the vehicle. This is usually connected to the throttle channel of a pistol grip transmitter, or the elevator channel of a dual stick transmitter. The R/C signal fed to the S2 input controls the turning of the vehicle.
When switch 4 is in the DOWN position, Independent mode is selected. In this mode, the signal fed to the S1 input directly controls Motor 1 (M1A and M1B) and the signal fed to S2 controls Motor 2.

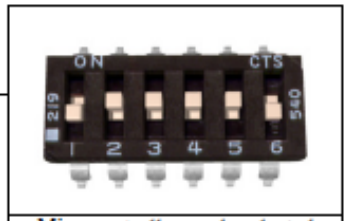
Switch 5: Exponential response

If switch 5 is in the UP position, the response is linear.
If switch 5 is in the DOWN position, the response to input signals will be exponential. This softens control around the zero speed point, which is useful for control of vehicles with fast top speeds or fast max turning rates.

Exponential mode enabled

Switch 6: R/C Mode/Microcontroller mode select

If switch 6 is in the UP position, then the Sabertooth is in standard R/C mode. This mode is designed to be used with a hobby-style transmitter and receiver. It automatically calibrates the control center and endpoints to maximize stick usage. It also enables a Timeout Failsafe, which will shut down the motors if the Sabertooth stops receiving correct signals from the receiver.



Microcontroller mode selected

If switch 6 is set in the DOWN position, then Microcontroller mode is enabled. This disables the Timeout Failsafe and auto-calibration. This means that the Sabertooth will continue to drive the motor according to the last command until another command is given. If the control link is possible unreliable – like a radio - then this can be dangerous due to the robot not stopping. However, it is extremely convenient if you are controlling the Sabertooth from a microcontroller. In this case, commanding the controller can be done with as little as three lines of code.

Output_High(Pin connected to S1)
Delay(1000us to 2000us)
Output_Low(Pin connected to S1)

A note on certain microprocessor receivers

Some receivers, such as the Spektrum AR6000, will output servo pulses before a valid transmitter signal is present. This will cause the Sabertooth to autocalibrate to the receiver's startup position which may not correspond to the center stick position, depending on trim settings. This may cause the motors to move slowly, even when the transmitter stick is centered. If you encounter this, either consult your receiver manual to reprogram the startup position, or adjust your transmitter trims until the motors stop moving. As a last resort, you can enter R/C microcontroller mode which will disable Sabertooth's autocalibration.

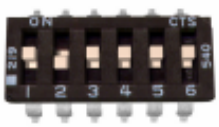
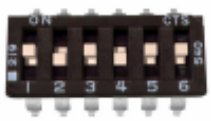
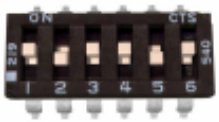
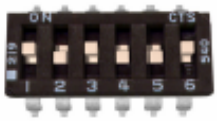
Mode 3: Simplified Serial Mode

Simplified serial uses TTL level single-byte serial commands to set the motor speed and direction. This makes it easy to interface to microcontrollers and PCs, without having to implement a packet-based communications protocol. Simplified serial is a one-direction only interface. The transmit line from the host is connected to S1. The host's receive line is not connected to the Sabertooth. Because of this, multiple drivers can be connected to the same serial transmitter. If using a true RS-232 device like a PC's serial port, it is necessary to use a level converter to shift the –10V to 10V rs-232 levels to the 0v-5v TTL levels the Sabertooth is expecting. This is usually done with a Max232 type chip. If using a TTL serial device like a microcontroller, the TX line of the microcontroller may be connected directly to S1.

Because Sabertooth controls two motors with one 8 byte character, when operating in Simplified Serial mode, each motor has 7 bits of resolution. Sending a character between 1 and 127 will control motor 1. 1 is full reverse, 64 is stop and 127 is full forward. Sending a character between 128 and 255 will control motor 2. 128 is full reverse, 192 is stop and 255 is full forward. Character 0 (hex 0x00) is a special case. Sending this character will shut down both motors.

Baud Rate Selection

Simplified Serial operates with an 8N1 protocol – 8 data bytes, no parity bits and one stop bit. The baud rate is selected by switches 4 and 5 from the following 4 options

	
2400 Baud: 01x00x	9600 Baud: 01x10x
	
19200 Baud: 01x01x	38400 Baud: 01x11x

What baud rate to use is dependent on what your host can provide and the update speed necessary. 9600 baud or 19200 baud is recommended as the best starting points. If communication is unreliable, decrease the baud rate. If communications are reliable, you may increase the baud rate. The maximum update speed on the Sabertooth is approximately 2000 commands per second. Sending characters faster than this will not cause problems, but it will not increase the responsiveness of the controller either.

The baud rate may be changed with power on by changing the DIP switch settings. There is no need to reset or cycle power after a baud rate change.

There are 2 operating options for Simplified Serial. These are selected by the position of Switch 6.

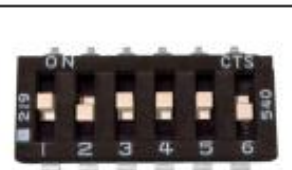
Option 1: Standard Simplified Serial Mode

Serial data is sent to input S1. The baud rate is selected with switches 4 and 5. Commands are sent as single bytes. Sending a value of 1-127 will command motor 1 Sending a value of 128-255 will command motor 2. Sending a value of 0 will shut down both motors.

Standard Simplified Serial

Option 2: Simplified Serial with Slave Select

This mode is used when it is desirable to have multiple Sabertooth motor drivers running from the same serial transmitter, but you do not wish to use packetized serial. A digital signal (0v or 5v) is fed to the S2 input. This is controlled by the host microcontroller. If the signal on S2 is logic high (5v) when the serial command is sent, then the driver will change to the new speed. If the signal on S2 is not high when the command is sent, then command will be ignored. Pseudo-code demonstrating this is shown below. After sending the signal, allow about 50 us before commanding the Slave Select line to a logic LOW to allow time for processing. A hookup diagram and example pseudo-code are shown in **Figures 6.2** and **6.3**.



Simplified Serial with Slave Select

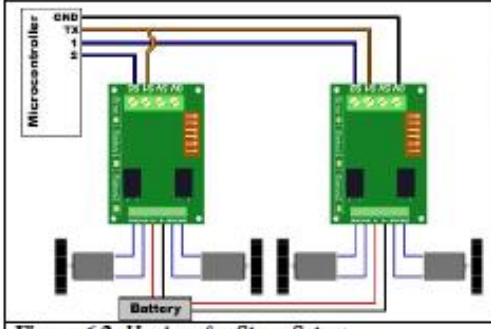


Figure 6.2: Hookup for Slave Select

```
//set controller 1's speed
Output_High (S2 pin on controller 1)
USART_TX(controller 1 speed, 0 to 255)
Delay_us(50)
Output_Low (S2 pin on controller 1)

//set controller 2's speed
Output_High (S2 pin on controller 2)
USART_TX(controller 2 speed, 0 to 255)
Delay_us(50)
Output_Low (S2 pin on controller 2)
```

Figure 6.3: Pseudocode for Slave Select

Mode 4: Packetized Serial Mode

Packetized Serial uses TTL level multi-byte serial commands to set the motor speed and direction. Packetized serial is a one-direction only interface. The transmit line from the host is connected to S1. The host's receive line is not connected to the Sabertooth. Because of this, multiple Sabertooth 2x10 motor drivers can be connected to the same serial transmitter. It is also possible to use SyRen and Sabertooth motor drivers together from the same serial source, as well as any other serial device, as long as it will not act on the packets sent to the Sabertooth. If using a true RS-232 device like a PC's serial port, it is necessary to use a level converter to shift the –10V to 10V rs-232 levels to the 0v-5v TTL. Packetized serial uses an address byte to select the target device. The baud rate is selected automatically by sending the bauding character (170 in decimal, AA in hex) before any commands are sent.

Packet Overview

The packet format for the Sabertooth consists of an address byte, a command byte, a data byte and a seven bit checksum. Address bytes have value greater than 128, and all subsequent bytes have values 127 or lower. This allows multiple types of devices to share the same serial line.

An example packet and pseudo-code to generate it are shown in **Figures 7.1** and **7.2**

Packet
Address: 130
Command : 0
Data: 64
Checksum: 66
	Void DriveForward(char address, char speed)
	{
	Putc(address);
	Putc(0);
	Putc(speed);
	Putc((address + 0 + speed) & 0b01111111);
	}

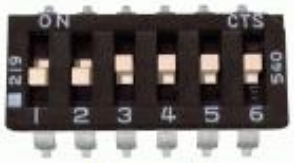
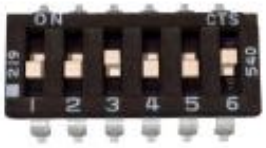
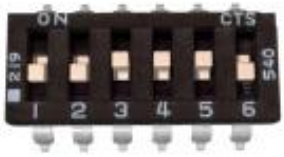
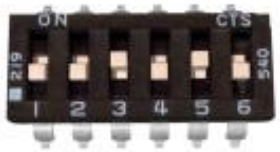
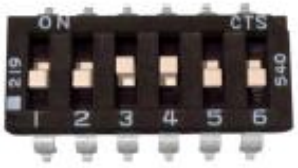
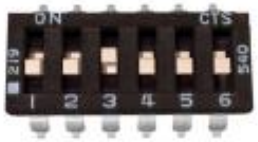
Figure 7.1: Example 50% forward	Figure 7.2: Pseudocode to generate 7.1

Baud Rate Selection:

Packetized Serial operates with an 8N1 protocol – 8 data bytes, no parity bits and one stop bit. The baud rate is automatically calculated by the first character sent. This character must be (170 in decimal) (binary 10101010) and must be sent before any serial communications are done. It is not possible to change the baud rate once the bauding character has been sent. The valid baud rates are 2400, 9600, 19200 and 38400 baud. Until the bauding character is sent, the driver will accept no commands and the green status1 light will stay lit. Please note that Sabertooth may take up to a second to start up after power is applied, depending on the power source being used. Sending the bauding character during this time period may cause undesirable results. **When using Packetized Serial mode, please allow a two-second delay between applying power and sending the bauding character to the drivers.**

Address Byte Configuration:

Address bytes are set by switches 4, 5 and 6. Addresses start at 128 and go to 135. The switch settings for the addresses are shown in the chart below

	
Address: 128	Address: 129
	
Address: 130	Address: 131
	
Address: 132	Address: 133
	
Address: 134	Address: 135

Commands:

The command byte is the second byte of the packet. There are four possible commands in packetized serial mode. Each is followed by one byte of data

0: Drive forward motor 1 (decimal 0, binary 0b00000000, hex 0h00)
This is used to command motor 1 to drive forward. Valid data is 0-127 for off to full forward drive. If a command of 0 is given, the Sabertooth will go into power save mode for motor 1 after approximately 4 seconds.

1: Drive backwards motor 1 (decimal 1, binary 0b00000001, hex 0h01)
This is used to command motor 1 to drive backwards. Valid data is 0-127 for off to full reverse drive. If a command of 0 is given, Sabertooth will go into power save mode for motor 1 after approximately 4 seconds.

2: Min voltage (decimal 2, binary 0b00000010, hex 0h02)
This is used to set a custom minimum voltage for the battery feeding the Sabertooth. If the battery voltage drops below this value, the output will shut down. This value is cleared at startup, so much be set each run. The value is sent in .2 volt increments with a command of zero corresponding to 6v, which is the minimum. Valid data is from 0 to 120. The function for converting volts to command data is

Value = (desired volts-6) x 5

3: Max voltage (decimal 3, binary 0b0000011, hex 0h03)
This is used to set a custom maximum voltage. If you are using a power supply that cannot sink current such as an ATX supply, the input voltage will rise when the driver is regenerating (slowing down the motor) Many ATX type supplies will shut down if the output voltage on the 12v supply rises beyond 16v. If the driver detects an input voltage above the set limit, it will put the motor into a hard brake until the voltage drops below the set point again. This is inefficient, because the energy is heating the motor instead of recharging a battery, but may be necessary. The driver comes preset for a maximum voltage of 30V. The range for a custom maximum voltage is 0v-25v. The formula for setting a custom maximum voltage is

Value = Desired Volts*5.12

If you are using any sort of battery, then this is not a problem and the max voltage should be left at the startup default.

4: Drive forward motor 2 (decimal 4, binary 0b00000100, hex 0h04)
This is used to command motor 2 to drive forward. Valid data is 0-127 for off to full forward drive. If a command of 0 is given, the Sabertooth will go into power save mode for motor 2 after approximately 4 seconds.

5: Drive backwards motor 2 (decimal 5, binary 0b00000101, hex 0h05)
This is used to command motor 2 to drive backwards. Valid data is 0-127 for off to full reverse drive. If a command of 0 is given, the Sabertooth will go into power save mode after approximately 4 seconds.

6: Drive motor 1 7 bit (decimal 6, binary 0b00000110, hex 0h06)
This command is used to drive motor 1. Instead of the standard commands 0 and 1, this one command can be used to drive motor 1 forward or in reverse, at a cost of lower resolution. A command of 0 will correspond to full reverse, and a command of 127 will command the motor to drive full forward. A command of 64 will stop the motor.

7: Drive motor 2 7 bit (decimal 7, binary 0b00000111, hex 0h07)
This command is used to drive motor 2. Instead of the standard commands 4 and 5, this one command can be used to drive motor 1 forward or in reverse, at a cost of lower resolution. A command of 0 will correspond to full reverse, and a command of 127 will command the motor to drive full forward. A command of 64 will stop the motor.

Mixed mode commands:

Sabertooth can also be sent mixed drive and turn commands. When using the mixed mode commands, please note that the Sabertooth requires valid data for both drive and turn before it will begin to operate. Once data for both has been sent, then each may be updated as needed, it is not necessary to send both data packets each time you with to update the speed or direction. You should design your code to either use the independent or the mixed commands. Switching between the command sets will cause the vehicle to stop until new data is sent for both motors.

8: Drive forward mixed mode (decimal 8, binary 0b00001000, hex 0h08)
This is used to command the vehicle to drive forward in mixed mode. Valid data is 0-127 for off to full forward drive.

9: Drive backwards mixed mode (decimal 9, binary 0b00001001, hex 0h09)
This is used to command the vehicle to drive backwards in mixed mode. Valid data is 0-127 for off to full reverse drive.

10: Turn right mixed mode (decimal 10, binary 0b00001010, hex 0h0a)
This is used to command the vehicle to turn right in mixed mode. Valid data is 0-127 for zero to maximum turning speed.

11: Drive turn left mixed mode (decimal 11, binary 0b00001011, hex 0h0b)
This is used to command the vehicle to turn leftt in mixed mode. Valid data is 0-127 for zero to maximum turning speed.

12: Drive forwards/back 7 bit (decimal 12, binary 0b00001100, hex 0h0c)
This is used to command the vehicle to move forwards or backwards. A command of 0 will cause maximum reverse, 64 will cause the vehicle to stop, and 127 will command full forward.

13: Turn 7 bit (decimal 13, binary 0b00001101, hex 0h0d)
This is used to command the vehicle turn right or left. A command of 0 will cause maximum left turn rate, 64 will cause the vehicle to stop turning , and 127 will command maximum right turn rate.

Checksum:

To prevent data corruption, each packet is terminated with a checksum. If the checksum is not correct, the data packet will not be acted upon. The checksum is calculated as follows:

Checksum = address byte +command byte +data byte

The checksum should be added with all unsigned 8 bit integers, and then ANDed with the mask 0b01111111 in an 8 bit system.

Example of Packetized Serial:

The following is an example function for commanding two Dimension Engineering motor drivers using Packetized Serial Mode. **Figure 7.3** shows an example hookup and **Figure 7.4** shows an example function.

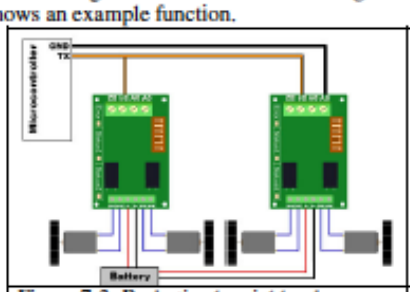


Figure 7.3: Packetized serial hookup

```
Void DriveForward(char address, char speed)
{
  Putc(address);
  Putc(0);
  Putc(speed);
  Putc((address + 0 + speed) & 0b01111111);
}
```

Figure 7.4: Packetized Serial Function

Example: So in this function, if address is 130, command is 0 (for driving forward), speed is 64, the checksum should calculate as follows:
130+0+64 = 194
194 in binary is 0b11000010
0b11000010 & 0b01111111 = 0b01000010

Once all the data is sent, this will result in the Sabertooth with address 130 driving forward at roughly half throttle.

Emergency Stop:

In Packetized Serial mode, the S2 input is configured as an active-low emergency stop. It is pulled high internally, so if this feature isn't needed, it can be ignored. If an emergency stop is desired, all the S2 inputs can be tied together. Pulling the S2 input low will cause the driver to shut down. This should be tied to an emergency stop button if used in a device that could endanger humans.