

## **Lab 1 An Autonomous Train Controller 3**

- 1.1 The Train Architecture 3
  - 1.1.1 Power and Drive Motor Control 3
  - 1.1.2 IR Sensor and Servo Interface 4
  - 1.1.3 PIC/CPLD 4
- 1.2 Lab Exercises 8
  - 1.2.1 Using the IR Range Sensor 9
  - 1.2.2 A quick and dirty PWM generator 9
  - 1.2.3 Putting it all together 11

## **Lab 2 Using the Servo Motor 12**

- 2.1 Servo motor basics 12
  - 2.1.1 Driving the Servo Motor 13
  - 2.1.2 Where is it pointing right now? 14
- 2.2 Using the Servo/IR Sensor 14

Indiana University  
Computer Science P442/542  
Laboratory Experiments

Steven D. Johnson, Caleb Hess, and Bryce Himebaugh

Copyright ©2000 Indiana University Computer Science Department

**DRAFT March 8, 2002**

*sjohnson@cs.indiana.edu*

# Laboratory 1 An Autonomous Train Controller

The goal of this project is to implement a control algorithm for an autonomous model railroad locomotive. The primary requirements for the control algorithm are:

- 1) It must allow two or more locomotives to share a track layout without colliding with each other or running into stationary obstructions. The track layout will include at least one crossing.
- 2) It must prevent deadlocking. Two locomotives must not stop, each waiting for the other to move first. In the case of a stationary obstruction, the locomotive should reverse direction.

A few other requirements are imposed to limit damage from collisions. In the event of a forward-moving collision, the locomotive must disable itself until a manual reset is performed. When moving in reverse, low speed must be used because there is no means to detect an obstruction other than mechanical contact.

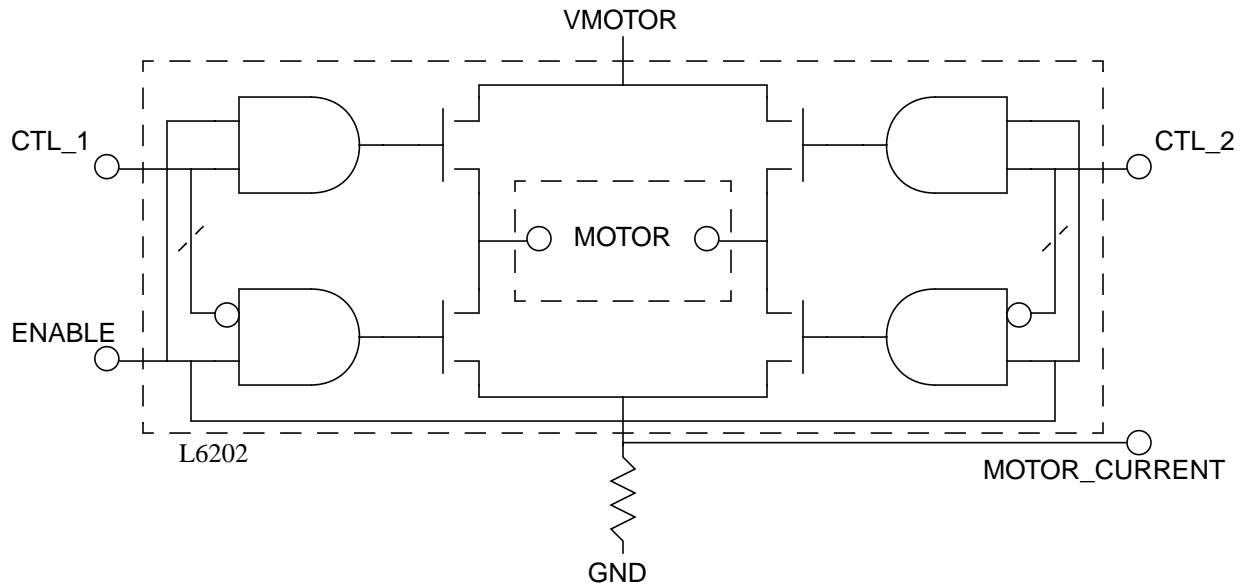
## 1.1 The Train Architecture

Each locomotive is built on a stock HO scale undercarriage. Minor wiring modifications allow onboard control logic to take power from the track and control the drive motor. An infrared object detector mounted on a servo motor provides range and bearing for objects in front of the train. The control algorithm resides in a PIC 16F877 and an XC9572 CPLD.

### 1.1.1 Power and Drive Motor Control

Refer to the [Power/Motor Schematic](#) drawing. 12 volt power is provided either from the track, or from a power supply connector (J5) for testing. 5 volt power (VCC) for the PIC, CPLD, IR sensor and servo motor comes from regulator U3. Since power pickup from the track through the wheels is subject to significant noise and dropouts, a backup 9 volt battery is used to ensure a clean 5 volt supply. The battery can also be used to power the logic for testing, but this is discouraged because the IR sensor and servo motor will quickly exhaust the battery. Please REMEMBER TO TURN OFF SWITCH S2 before storing your locomotive between sessions. The unregulated voltage VOP is used to power several op amps described below.

The drive motor is powered by 12 volts DC through switch S1 (VMOTOR). U4 is an H-bridge motor controller (Fig. 1) that allows control logic to switch the motor on and off and reverse its direction. MOTOR\_CONTROL\_1 and MOTOR\_CONTROL\_2 are used to control direction by turning on a diagonal pair of transistors - 10 will turn on the top left and lower right transistors, causing current to flow through the motor from left to right, while 01 turns on the opposite pair of transistors, causing current to flow through the motor from right to left. MOTOR\_ENABLE provides overall on/off control by enabling or disabling all four transistors.



**Figure 1-1 L6202 H-Bridge Motor Controller**

A feedback signal from the motor controller allows you to monitor motor current. The resistor between the controller chip and ground produces a small voltage which is proportional to motor current and which can be measured by an analog-to-digital converter on the PIC. This signal, combined with the motor drive control signals, may be used to obtain speed and acceleration data.

### 1.1.2 IR Sensor and Servo Interface

An IR rangefinder is used to detect objects in front of the train. It produces an analog voltage output to indicate the distance to an object, ranging from less than 0.5 V for distances greater than about 30 inches up to a maximum of 2.5 V at about 6 inches. The response is nonlinear, with most of the voltage range (roughly 1 to 2.5 V) corresponding to the nearest 10 or 12 inches. The IR sensor has one unfortunate limitation, in that its output peaks at a range of about 6 inches and then drops sharply, so you will want to ensure that nothing ever gets closer than that threshold!

The IR sensor's output is scaled through an op amp (See [IR Scanner Schematic](#) drawing) to expand its range to 0 to 5 V for feeding an analog-to-digital converter input on the PIC. Another op amp drives an LED which gives you a visual indication of the IR sensor's output.

A servo motor allows the IR sensor to scan an area of about 180 degrees in front of the train. The servo motor can be told to position itself at a given angle of rotation, but being mechanical, it takes some interval of time for it to get there. To allow your controller to know where the sensor is pointing at any moment, an analog position voltage is provided. Again, this voltage is scaled by an op amp and is fed to one of the PIC's A/D converter inputs. You will need to empirically determine the voltage/angle relationship.

### 1.1.3 PIC/CPLD

Your control logic will be implemented in a PIC 16F877 extended by an XC9572 CPLD. While the PIC alone might be sufficient, the addition of some programmable logic allows the problem to

naturally divide into a sequential control algorithm running on the PIC, and various peripheral functions residing in the CPLD. For example, the safety interlock logic to disable the drive motor in a collision should be implemented in the CPLD and should function independently without requiring attention from the PIC. A modular approach such as this offers both improved reliability and reduced overall complexity.

The PIC and CPLD have been prewired to each other and to the various hardware functions as shown in the PIC/CPLD Schematic. In addition to the PIC and CPLD chips, the schematic shows a 20 MHz oscillator module, an RS-232 interface, front and rear collision detect switches, a manual reset button, and an 8 bit DIP switch. Connector J9 and the Servo Index sensor are not implemented. J10 is a ribbon cable header to allow the circuit board to be used for other projects.

The following two tables list the available functions assigned to each pin of the PIC and CPLD.

**Table 1-1**

<i>PIC PIN ASSIGNMENTS</i>				
<i>PIC PIN #</i>	<i>PIN NAME</i>	<i>PIC FUNCTION</i>	<i>PREWIRED USE</i>	<i>CPLD PIN #</i>
14	CLKIN	External clock input	20 MHz. oscillator input	
15	CLKOUT	Clock buffer output	20 MHz to CPLD GCK1	5
2	$\overline{\text{MCLR}}/V_{pp}$	Clear / programming voltage	ICD programmer, pin 6	
3	RA0/AN0	Analog input 0	IR range data input	
4	RA1/AN1	Analog input 1	Servo position input	
5	RA2	Digital I/O	CPLD GTS2	40
6	RA3/AN3	Analog input 2	Motor current sense input	
7	RA4/T0CKI	Digital I/O, Timer 0 clock input	CPLD	1
8	RA5	Digital I/O	DIPSW 1 CPLD	24
36	RB0/INT	Digital I/O, External Interrupt	CPLD	9
37	RB1	Digital I/O	DIPSW 2 CPLD	8
38	RB2	Digital I/O	DIPSW 3 CPLD	4
39	RB3/PGM	Low voltage programming input	ICD programmer, pin 1	
41	RB4/INT	Digital I/O, Interrupt on change	DIPSW 4 CPLD	3
42	RB5/INT	Digital I/O, Interrupt on change	DIPSW 5 CPLD	2
43	RB6/PGC	Serial programming clock	ICD programmer, pin 2	
44	RB7/PGD	Serial programming data	ICD programmer, pin 3	
16	RC0/T1CKI	Digital I/O, Timer 1 clock input	CPLD	28

**Table 1-1**

<i>PIC PIN ASSIGNMENTS</i>				
<i>PIC PIN #</i>	<i>PIN NAME</i>	<i>PIC FUNCTION</i>	<i>PREWIRED USE</i>	<i>CPLD PIN #</i>
18	RC1/CCP2	Digital I/O, Capture/Compare/PWM2	CPLD GCK3	7
19	RC2/CCP1	Digital I/O, Capture/Compare/PWM1	CPLD GCK2	6
20	RC3	Digital I/O	DIPSW 6 CPLD	27
25	RC4	Digital I/O	DIPSW 7 CPLD	26
26	RC5	Digital I/O	DIPSW 8 CPLD	25
27	RC6/TX	USART Transmit	Serial I/O Transmit	
29	RC7/RX	USART Receive	Serial I/O Receive	
21	RD0/PSP0	Digital I/O	CPLD	22
22	RD1/PSP1	Digital I/O	CPLD	20
23	RD2/PSP2	Digital I/O	CPLD	19
24	RD3/PSP3	Digital I/O	CPLD	18
30	RD4/PSP4	Digital I/O	CPLD	14
31	RD5/PSP5	Digital I/O	CPLD	13
32	RD6/PSP6	Digital I/O	CPLD	12
33	RD7/PSP7	Digital I/O	CPLD	11
9	RE0/RD	Digital I/O, parallel port READ	CPLD	44
10	RE1/ $\overline{WR}$	Digital I/O, parallel port WRITE	CPLD	43
11	RE2/ $\overline{CS}$	Digital I/O, parallel port SELECT	CPLD GTS1	42
13, 34		Vss	GND	10, 23, 31
12, 35		Vdd	+5 V	21, 32, 41
1, 17, 28, 40		NC		

**Table 1-2**

<i>CPLD PIN ASSIGNMENTS</i>						
<i>CPLD PIN #</i>	<i>PIN NAME</i>		<i>CPLD FUNCTION</i>	<i>PREWIRED USE</i>		<i>PIC PIN #</i>
	<i>FB</i>	<i>CELL</i>				
1	1	2	I/O	J10-3	RA4/TOCKI	7
2	1	5	I/O	DIPSW5	RB5/INT	42
3	1	6	I/O	DIPSW4	RB4/INT	41
4	1	8	I/O	DIPSW3	RB2	38
5	1	9	I/O, GCK1 - Global Clock 1	20 MHz Clock buffered thru PIC		15
6	1	11	I/O, GCK2 - Global Clock 2	RC2/CCP1		19
7	1	14	I/O, GCK3 - Global Clock 3	RC1/CCP2		18
8	1	15	I/O	DIPSW2	RB1	37
9	1	17	I/O	RB0/INT		36
10			GND			
11	3	2	I/O	RD7		33
12	3	5	I/O	RD6		32
13	3	8	I/O	RD5		31
14	3	9	I/O	RD4		30
15			TDI	XChecker Header		
16			TMS			
17			TCK			
18	3	11	I/O	RD3		24
19	3	14	I/O	RD2		23
20	3	15	I/O	RD1		22
21			VccInternal	+5 V		
22	3	17	I/O	RD0		21
23			GND			
24	4	2	I/O	DIPSW1	RA5	8
25	4	5	I/O	DIPSW8	RC5	26
26	4	8	I/O	DIPSW7	RC4	25

**Table 1-2**

<i>CPLD PIN ASSIGNMENTS</i>						
<i>CPLD PIN #</i>	<i>PIN NAME</i>		<i>CPLD FUNCTION</i>	<i>PREWIRED USE</i>		<i>PIC PIN #</i>
	<i>FB</i>	<i>CELL</i>				
27	4	9	I/O	DIPSW6	RC3	20
28	4	11	I/O		RC0	16
29	4	14	I/O	J10-13 Servo Index (unimplemented)		
30			TDO	XChecker Header		
31			GND			
32			VccIO	+5 V		
33	4	15	I/O	J10-12	Rear Collision Det. IN	
34	4	17	I/O	J10-6	Servo PWM OUT	
35	2	2	I/O	J10-7	Motor Control 1 OUT	
36	2	5	I/O	J10-8	Motor Control 2 OUT	
37	2	6	I/O	J10-9	Motor Enable OUT	
38	2	8	I/O	J10-11	Front Collision Det. IN	
39	2	9	I/O, GSR - Global Set/Reset		$\overline{\text{Reset}}$ IN	
40	2	11	I/O, GTS2 - Global TriState 2	J10-4	RA2	5
41			VccInternal	+ 5 V		
42	2	14	I/O, GTS1 - Global TriState 1		RE2	11
43	2	15	I/O		RE1	10
44	2	17	I/O		RE0	9

## 1.2 Lab Exercises

At this stage, you will be working with your train on the bench, so you will need to provide power other than from the track. Plug in your bench power supply to the 4-pin connector on the rear corner of the board (you may need an adapter). The green LED beside the power connector will light when power is applied. Turn on the Logic power switch and the clear LED4 should light to indicate that 5 Volt power is available. Remember to turn this switch off when disconnecting the Bench Power supply, to avoid draining the backup battery. Keep the Motor switch off unless you are testing the drive motor with the wheels raised off the bench! Plug in your PIC ICD module to the connector under the reset button, and hook up your Xilinx XChecker cable to the header beside the DIP switch.

### 1.2.1 Using the IR Range Sensor

With 5 volt power turned on, look at the red LED in front of the servo motor. Move your hand back and forth in front of the IR sensor and the LED should brighten and dim. This gives you a visual indication of the sensor's output. In order to use this data, you will need to program the PIC's Analog-to-Digital converter to provide a numeric value for the distance to an object. The A/D converter is described in section 11 of the PIC data sheet.

To use the A/D converter, you will need to configure the following registers:

TRISA - set to binary 001011 to make RA0/AN0, RA1/AN1, RA3/AN3 inputs

ADCON1 - set to 0x04 to use RA0, RA1, RA3 as analog inputs

ADCON0 - set to 0x81 (Fosc/32, input channel 0, A/D converter on).

In addition, you must provide an acquisition time of at least 20 microseconds before starting the A/D conversion. A convenient way to do this is by using CCP2's event trigger, which will start an A/D conversion after a delay determined by TMR1. To use TMR1 and CCP2, configure these registers:

CCPR2 - clear CCPR2H, set CCPR2L to 0x64 (5 MHz internal clock / 100 => 20 uS)

CCP2CON - set to 0x0B for compare mode with event trigger

TMR1 - clear TMR1H, TMR1L

T1CON - set to 0x01 to start timing.

Now loop, testing ADCON bit 2, until TMR1/CCP2 sets it to start the A/D conversion. Then loop, again testing ADCON bit 2, until the A/D converter clears it to indicate that the conversion is complete. Nest these two loops inside another loop. Run your program on the PIC while monitoring ADRESH and ADRESL in a watch window with objects at various distances in front of the IR sensor.

The IR sensor's output tends to be quite noisy. How many bits of ADRES are actually useful?

### 1.2.2 A quick and dirty Pulse Width Modulator

Our drive motor is a simple permanent magnet DC motor whose speed is proportional to the amount of current flowing through it. In the analog world, we might control its speed using a continuously variable resistor to regulate the current flow. In the digital world, we deal only with ON and OFF conditions, so we have to be a bit creative in order to simulate analog speed control. The technique we will use consists of rapidly switching the motor on and off for varying time periods, so that the *time-averaged* motor current can be varied in small increments. This technique of simulating analog output is called **Pulse Width Modulation**.

For various reasons, the PIC's onboard PWM is not well suited for driving our motors. We will, instead, build our own PWM using the CPLD. Our first problem is to divide down the 20 MHz oscillator to something more suitable for driving motors - a few hundred Hz. for the drive motor, and 50 Hz. for the servo. Looking at the resources available, one solution would be to use the PIC's CCP1 as a programmable divider to provide a slow clock to the CPLD. Since we're already

using CCP2 in compare mode for A/D timing, TMR1 is not available. We can, however, use CCP1 in PWM mode so as to access TMR2. TMR2 with its prescaler gives us 12 bits, so we could get a clock output as low as 1.22 KHz (internal 5 MHz clock / 4096).

To do this, configure the PWM as follows:

TRISC - RC2/CCP1 must be an output

CCPR1L - sets the duty cycle, must be less than the output period - 0x20 should be safe

T2CON - 0x05 turns on TMR2 with prescaler at 1:4. Prescaler affects PR2 calculation!

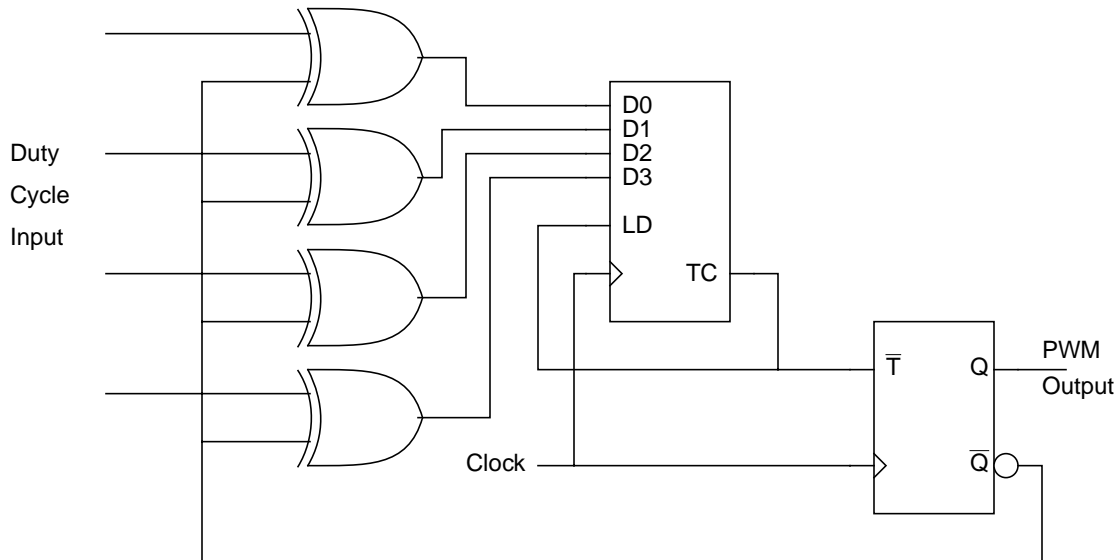
PR2 - clock divisor; calculate the value needed to get 10 KHz out, for now

CCP1CON - set to 0x0F for PWM mode.

Configure your PIC and check the CCP1 output with an oscilloscope. Adjust the value of PR2, if you are not getting 10 KHz.

Now that we have a reasonably slow clock, we can get on to the PWM itself. Pulse Width Modulation has two parts: the *period*, or rate at which pulses are generated, and the *duty cycle*, or length of each pulse relative to the period. We will need a counter of some sort to determine the period, and it should be obvious that the number of bits in the period counter will determine the degree of resolution, or number of different pulse widths, that can be produced. In the case of our train's drive motor, we don't need a lot of resolution. A four-bit counter will give us 16 speed increments, which should be more than sufficient.

To set the duty cycle, we could use a register and comparator as in the PIC (Fig. 8-3 in the datasheet). However, you might notice that the ON and OFF periods of the PWM waveform are complementary, if the period is fixed. In other words,  $T_{off} = T_{period} - T_{on}$ . This suggests that we might dispense with the comparator and simply use our counter's Terminal Count output, alternately counting from DutyCycle to TC and from -DutyCycle to TC. A full implementation of this would require carry logic, in order to complement and increment. If we are willing to accept a slight error, we can simply use the 1's complement instead of 2's complement. This approach leads to the following design:

**Figure 1-2 Four-bit PWM generator**

This design has two minor flaws. First, by not doing a complete binary negation, we add one count per cycle. The consequence of this is that the output pulse rate will be  $\text{Clock} / 2^n + 1$  instead of  $\text{Clock} / 2^n$ . Second, the Toggle flip-flop will be toggled twice in each cycle as the counter reaches TC, which means that we cannot produce a duty cycle of 0% or 100% - there will always be a one-clock-wide pulse. In our application, neither of these errors is significant.

Design a PWM for the drive motor and implement it in the CPLD. Using the 10 KHz clock at GCK2, what should your motor pulse rate be? Check the output with an oscilloscope at the top of R18. For initial testing, you may use the DIPSW to manually enter the desired duty cycle.

### 1.2.3 How do I stop it?

Our locomotives do not have brakes in the conventional sense. Simply turning off the drive motor will allow the locomotive to coast for some distance before stopping. There is, however, a way to obtain braking action from the motor, by utilizing the fact that it behaves as a generator when mechanical force is applied to its shaft. Turning on the **MOTOR\_ENABLE** signal and setting both **MOTOR\_CONTROL** signals to 0 will turn on the two lower transistors in the motor controller, effectively placing a short across the motor/generator. This quickly dissipates the locomotive's energy of motion in the form of heat in the motor windings and greatly reduces the stopping distance. For even faster stopping, you might try driving the motor in reverse. This is a bit risky, since you can't tell when the locomotive has stopped or begun moving backward.

### 1.2.4 Putting it all together

You now have enough to implement a simple system that should slow down, brake, and then stop

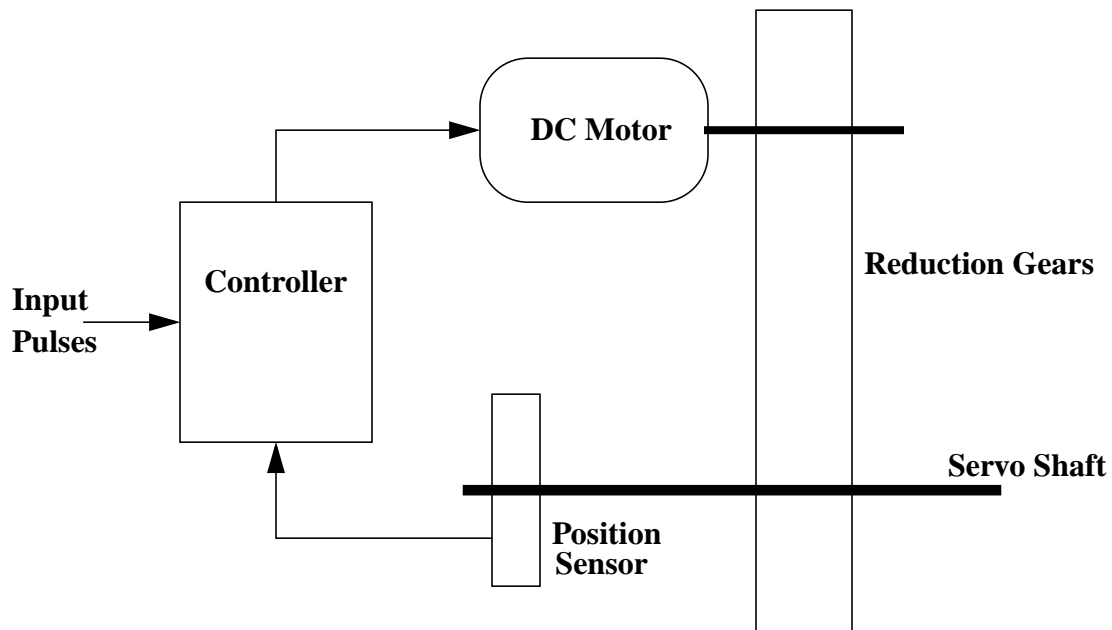
before colliding with a stationary obstruction. Set the `MOTOR_CONTROL[12]` signals to make the train run forward. Use your PWM output to control the drive motor's speed by pulsing `MOTOR_ENABLE` on and off. Add the front collision detect switch to your CPLD design so that a collision will latch a flip-flop and disable the drive motor until the reset button is pressed. Have the PIC set the motor speed based on the IR sensor's output. (A quick way to do this is to truncate `ADRES` to its 4 MSB, complement them, and send them directly to the PWM.) Try it out on the track!

## Laboratory 2 Using the Servo Motor

A Servo Motor is an angular positioning device. It can be instructed to move to a given angle of rotation, and will respond by turning until it reaches the desired position.

### 2.1 Servo motor basics

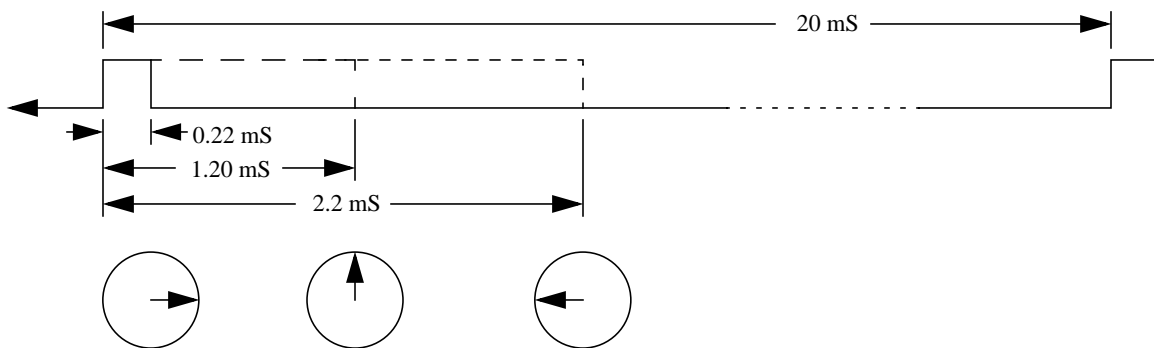
A servo motor has four parts, as shown in the drawing below: a simple DC motor, a reduction gear train, a position sensor, and a controller. The gear reduction ratio is quite large, so that many revolutions of the motor are required to move the servo shaft through just a few degrees. The position sensor is a potentiometer (variable resistor) mounted on the servo shaft. The controller produces internal pulses with a width determined by the resistance of the position sensor, and compares these pulses with a stream of externally supplied control pulses. Any difference between the internal and external pulse widths causes the motor to run for a time determined by the magnitude of the difference, and in a direction determined by the sign of the difference. Thus, the motor will position its shaft so that the pulses match, and will actively maintain that position until the external control pulses become longer or shorter.



**Figure 2-1 Servo Motor Components**

### 2.1.1 Driving the Servo Motor

Control pulses must be supplied to the servo continuously, as the motor is only energized when there is a difference between its internal position pulse and the external control pulse (in fact, the internal pulse is triggered by the rising edge of the external control pulse). If the control pulse rate is too slow, an external force will be able to move the servo shaft out of position between pulses, resulting in inaccurate positioning and vibration. On the other hand, the servo control logic needs some minimum amount of time in which to respond to each control pulse. In practice, servos of the sort we are using expect to receive a pulse about every 20 mS. Pulse width may vary from some small value (about 0.20 mS for our servos, as much as 0.8 mS for others) for full clockwise rotation to 2.4 mS for full counter-clockwise rotation. Most servos have a rotational range of about 210 degrees, with an intended working range of 180 degrees.



**Figure 2-2 Servo control pulse timing**

You can generate the servo control pulses using a PWM similar to the 4-bit PWM example of the previous lab. You will need to modify the design to limit its range of pulse widths to the range understood by the servo controller. For our use, we would like at least 20 discrete steps in the 180 degree range of rotation because the IR sensor's beam is about 10 degrees wide. Dividing 2.2 mS by 20, we find that the pulse width increment should be less than 0.11 mS. This means that we will need at least 182 counts to make up our 20 mS base period ( $20 \text{ mS} / 0.11 \text{ mS} = 182$ ). An 8 bit PWM counter will give us 257 counts in 20 mS, or .08 mS per count. This gives us 28 steps in 2.2 mS, or about 25 steps in the useful 180 degree range (the shortest 2 - 3 pulse widths will be too short), or about 7 degrees per step. So, we may use an 8 bit PWM counter to set the 20 mS base rate, but we will use only the lower 5 bits to specify a valid pulse width. The upper 3 input bits will always be zero, or ground (but remember, these bits must be complemented along with the 5 bits we actually use).

We can expect that the servo will spend much of its time sweeping from side to side, scanning for obstructions ahead. The PIC could perform this function, but we can also integrate it into the servo's PWM by adding a 5 bit loadable up/down counter between the PIC and the PWM counter. The library part CB8CLED is ideal for the purpose, providing count enable, up/down, and overriding load inputs. With a JK flip-flop for storing count direction plus two magnitude comparators (COMP8) for setting the upper and lower count limits we have a self-contained scanner. The load function allows the PIC to select a fixed angle when needed. One remaining detail is the rate

to scan at. The servo pulse itself is an obvious and convenient clock, however it makes the scan count a bit faster than the servo can keep up with - the motor will not reach the end of its range before the counter reverses direction. This may be acceptable, if you don't really need to sweep a full 180 degrees. Alternatively, you might divide down the servo pulse rate through another flip-flop if you want the motor's sweep to match the counter's range.

### **2.1.2 Where is it pointing right now?**

Being a mechanical device, the servo moves very slowly relative to CPU instruction execution. This presents a problem, in that when we tell the servo to move to some angle, we may have to wait an unknown number of instruction cycles before it reaches the assigned position. We have modified your servo motor by bringing out the position sensor signal and wiring it to A/D converter input AN1. This allows you to monitor the servo's actual position while it is moving, and to determine when it has stopped. The position sensor voltage at the PIC input varies from about 0.3 V fully clockwise to 4 V fully counterclockwise. As with the IR sensor's output, the analog signal will have some noise on it, and the lower 2-3 bits of the digital value will not be useful. You may want to build a table of position voltage vs. pulse width so that, for example, you could sweep the servo through its complete range, noting the position voltage anywhere an obstacle is detected. You would then be able to return to that angle by looking up the corresponding PWM pulse width input.

You might also want to provide the servo pulses to a PIC input, so that your control algorithm can determine, for example, when the scan counter has been clocked so that the servo is no longer scanning at random but is now moving toward a fixed position.

## **2.2 Using the Servo/IR Sensor**

You should now be able to design a control algorithm that will track and avoid colliding with one other moving train. You will first need to scan until you locate an object which is nearer than some threshold - maybe 30 cm., or about 2 volts IR input. Then move the IR beam toward the center until the object disappears. Keep the beam at this angle and wait for the object to reappear - if it does, and it is now closer, then reduce speed. Repeat until the object is moving away, as indicated by increasing distance. If it does not reappear after some time interval, resume scanning. If nothing is found nearer than your threshold, resume full speed.

Think about how the track layout affects your control algorithm. Are there possible track arrangements where the above procedure will fail? Can the algorithm as described deadlock? How can you prevent deadlock?