Micro-controllers for Model Railroads

Chapter 1: A Servo Controller using a PIC

Duncan McRee, Ph.D.

HOn3 Nevada County Narrow Gauge Railroad #9 delivers construction materials to the railhead across a turnout controlled by the Micro690 Servo Driver.

Introduction

When my article *Servos for Model Railroads* was published in MRH Issue #3, a number of readers responded positively to my offer to write more about using micro-controllers for model railroads. While this is probably not a topic that every modeler is interested in, for those that are interested in using micro-controllers, there is not a lot of material specifically aimed at model railroads available. Given the amazing value micro-controllers represent, a complete computer and I/O package on a chip costing a few bucks, these are amazingly useful devices and deserve some effort by anyone interested in modern electronics. Once mastered, they can be used for an amazing number of fun gadgets, replacing circuits that require dozens to hundreds of discrete components with a micro-controller and just a few parts.

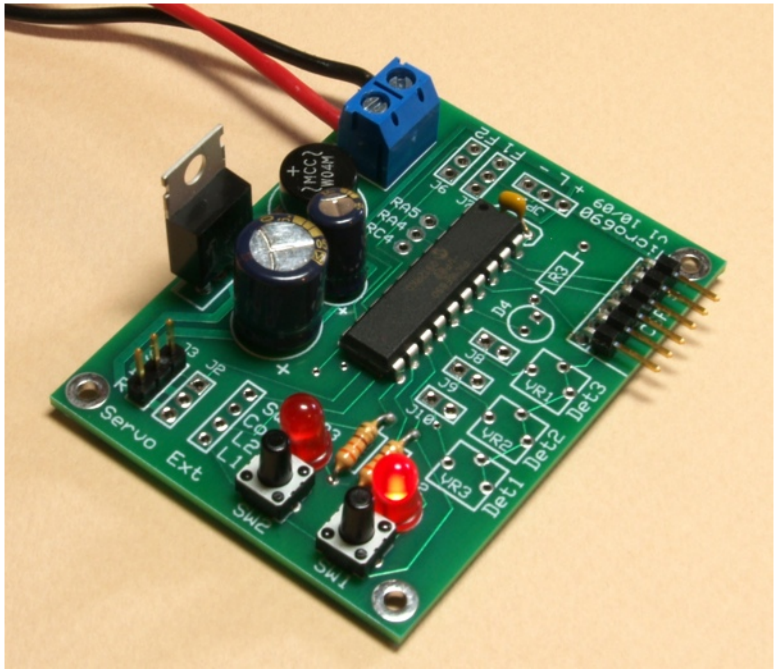


Fig. 1. The Micro690 Servo Driver. The power block is on the upper left with the wires attached, the servo connection is on the lower left corner and the programming header is on the upper right. The buttons and LED on the lower right are used for controlling and training the servo positions.

In this first chapter, we will make a basic servo controller that can be used to actuate a turnout or any other device that has two states such as a crossing gate, semaphore or ball signal. In the later articles we will add modules until eventually we will build a fully functional grade crossing guard circuit including flashing lights and train detection. A crossing guard is a good example project because it will cover driving servos to actuate the gate, flashing LEDs for the lights and detecting trains using photocells. The micro we will use is a PIC16F690 made by Microchip. This chip is an amazing bargain - it costs about $3 ($2 in quantities of 25) and is essentially a complete computer on a chip with a 4K programmable flash memory, an onboard oscillator and a built-in 256-byte EEPROM memory we can use for remembering things when the power is off. There are also several timers that will be used for sending the PWM signal to control the servo.

Implementing a micro-controller can be broken into two parts, hardware and software. First, let's cover the hardware. It is remarkably simple thanks to the micro-controller.

Hardware

The circuit is shown in Fig. 2. Let's break the circuit down into some of the sub-components. These components show up over and over again in micro-controller circuits.

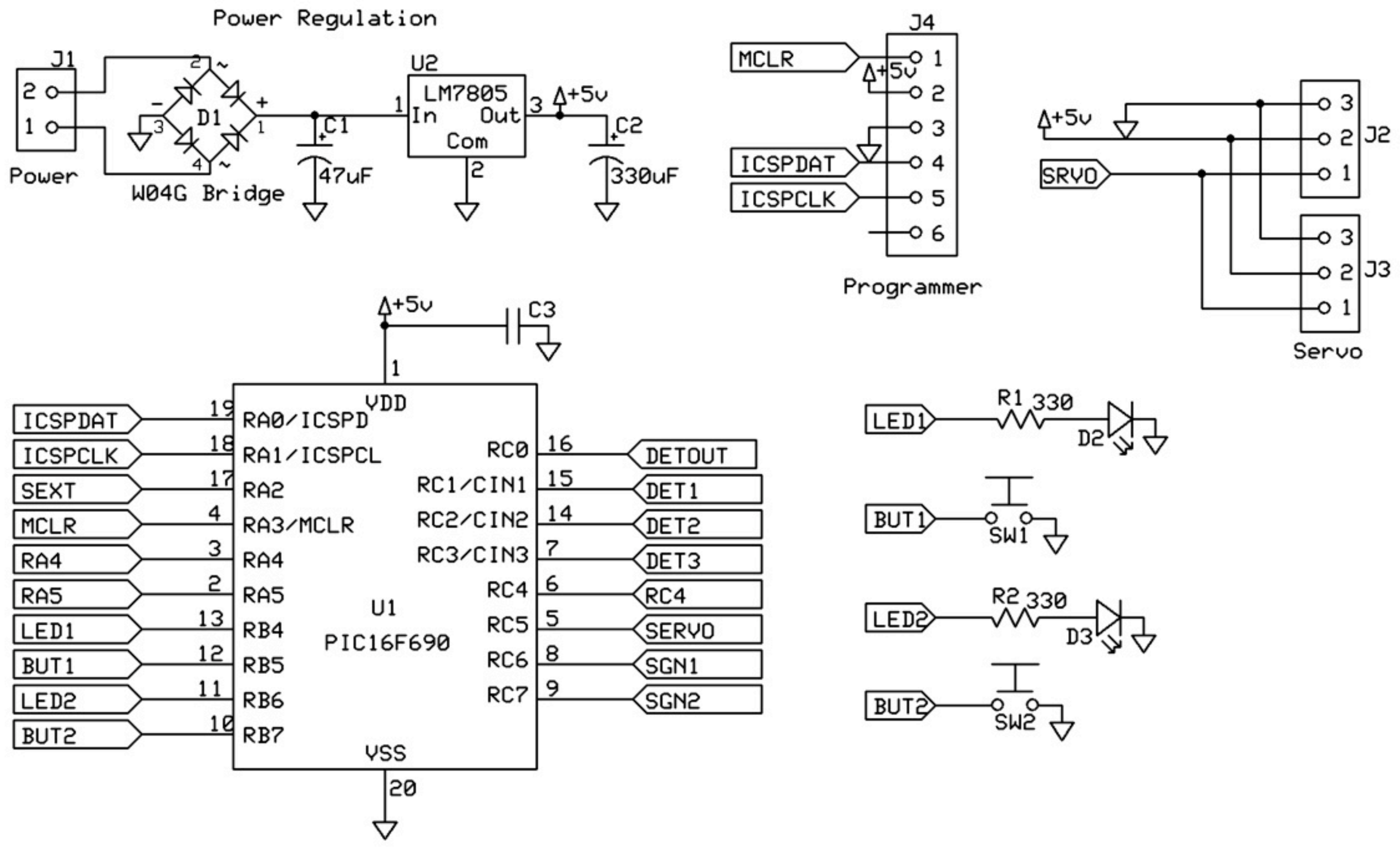


Fig. 2. Schematic for Micro690 Servo Driver.

Pins

The micro-controller communicates with the outside world via pins. The PIC16F690 has 20 pins of which 18 can be configured for I/O, RA0-RA5, RB4-RB7 and RC0-RC7. Since this is a digital device the pins talk in digital bits - which can be one of two values, "low", also called "0" and equal to 0 volts, or "high", also called "1", which is equal to 5 volts. Every pin on a micro-controller can be set as an output or as an input. As an output the pin can source or sink 25 mA of current. This enough to light several LEDs or drive a servo. As an input the pin can be used to read the value of a switch or other signal. The micro-controller runs at 4 MHz so it can be be used to read or generate very fast signals, for example DCC signals. This example of a servo controller will be a snap for the micro - it will spend 99% of its time idling.

LEDs

To light an LED all we need to do is connect a pin to the LED through a current-limiting resistor and configure that pin as an output. The basic circuit is shown in Fig. 3. If we set the pin to a 1, the LED will light and if we set the pin to 0 the LED will go out

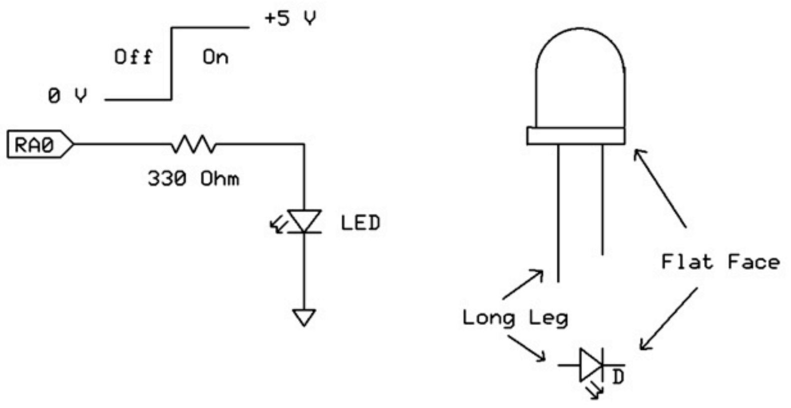


Fig. 3. Hooking up an LED to a micro.

.

Pushbuttons

Input from the user is often done through a momentary normally-open switch, which I will call a pushbutton, since the word “switch” has several other meanings in model railroading. The basic circuit is shown in Fig. 4. The pushbutton is connected across ground and a pull-up resistor. The pull-up resistor brings the input line up to +5V (1) when the button is not being pressed. When the pushbutton is pressed, and the contacts close, the input line is pulled to ground (0). In fact, the pull-up resistor is not needed if the micro-controller has internal pull-ups. The PIC16F690 has pull-ups on the RA and RB pins that can be turned on in software and we will take advantage of them in our design so that the resistor is not needed in the final schematic.

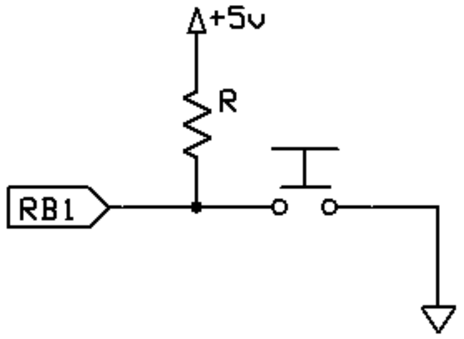


Fig. 4. Hooking up a pushbutton switch to a micro. When the pushbutton is open R will pull up the voltage at RB1 to +5V. When the pushbutton is closed RB1 will be pulled down to ground.

Using a pushbutton is a bit more complicated than this due to a phenomenon called “switch-bounce”. When the contacts are being closed the contacts don’t make a clean edge. Instead, the contacts bounce several times very quickly and make a brief series of 0 and 1’s before settling down. This can be taken care of by using some extra hardware or in software. We will of course do it in software since software is essentially “free”.

Servos

A servo has 3 input wires, signal, +5V and ground. To connect a servo to a micro-controller all we have to do is connect one of the output pins to the signal wire and the other two wires to the power supply. A 3-pin header is used on the board and the servo plugs directly in to this plug. See Fig. 5. The rest of the servo circuit is implemented in software.

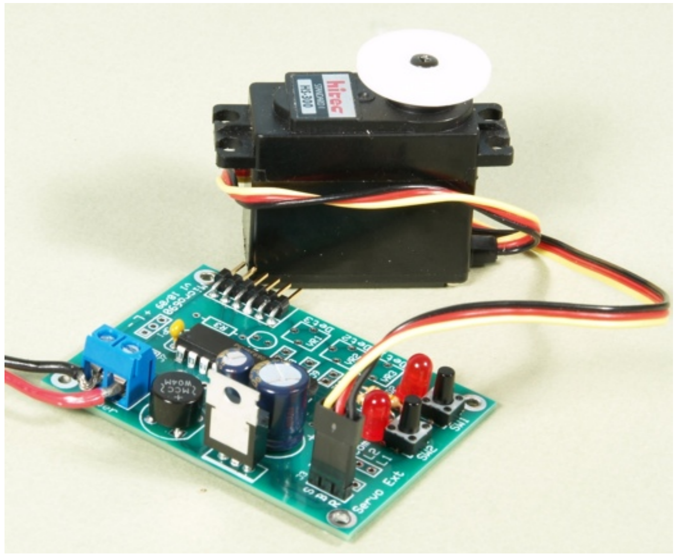
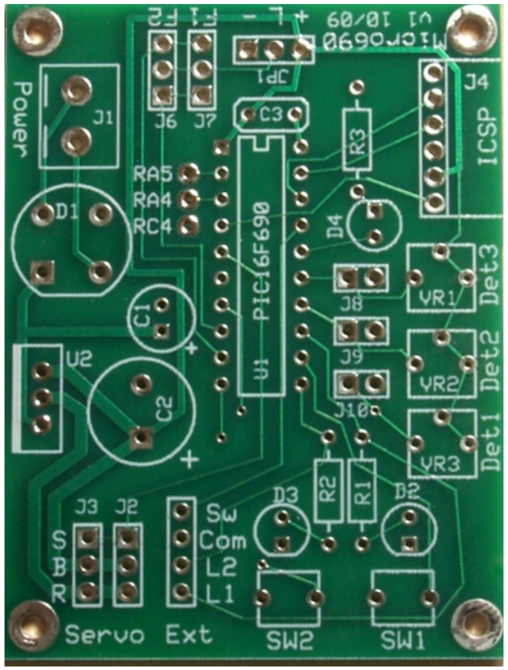


Fig. 5. The servo is attached to the board through a 3-pin header. Note that the yellow signal wire is attached to “S”. The signal wire may be yellow, white or brown depending upon the brand of servo. The other two wires carry power. (Unfortunately, there is an error on the PC board and the B and R markings are reversed).

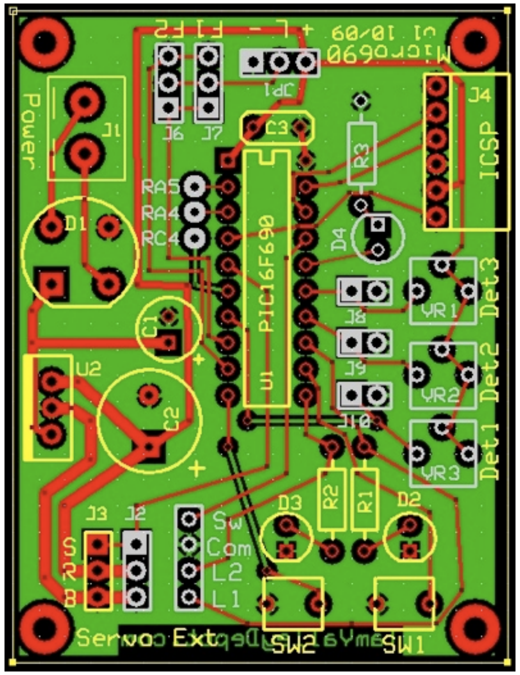
If two servos get the same input signal they will move in the same way. On the previous schematic, there are two servo ports wired in parallel. This will let you plug in two servos that will operate in tandem - i.e., one for each of two crossing gates on either side of a grade crossing.

PC Board - putting it all together

I have designed a PC board (Fig. 6) on which you can mount the components. I also give a Parts List with part numbers from Digi-Key in Table 1. The board has a number of unused pins and spots for extra components that we will use in future designs in this series. A programming header is included on the PC board that can be used to update the software later through a chip programmer (see **Programming the PIC** below). The parts not needed for this part of the series have been grayed out. The circuit could also be easily built on a breadboard or a piece of perf-board. You can use the PC board drawing as a wiring guide.



A



B

Fig. 6. **A) Micro690 PC board.** (Unfortunately, there is an error on the PC board and the B and R markings are reversed). **B) PC board layout.** The parts not needed for this project, a servo controller, have been grayed out. This same PC board will be used in all parts of the series and eventually become a fully functioning grade crossing control circuit including servos to operate the gates, flasher drive circuitry and optical train detection with phototransistors.

Table 1. Parts List for Micro690 Servo Driver

ID	Description	Digikey Part #
U1	PIC16F690 20-pin DIP	PIC16F690-I/P-ND
U2	LM7805 5V Regulator TO-220	LM7805CT-ND
D1	W04M 1.5A 400V Bridge Rectifier WOM	W04M-BPMS-ND
J1	5mm 2-pos Terminal Block	A97996-ND
J2, J3	3-pin 0.1” Male Header	S1011E-36-ND
J4	6-pin 0.1” Right-Angle Male Header	S1111E-36-ND
C1	100uF 25V Electrolytic Capacitor (value not critical)	P10269-ND
C2	1000uF 6.3V Electrolytic Capacitor (value not critical)	P10199-ND
C3	0.1 uF Capacitor (value not critical)	BC1114CT-ND
R1, R2	330 Ohm 1/4W Resistor	330QBK-ND
SW1, SW2	Pushbutton Switch 0.2” NO	P12231STB-ND
D2, D3	LED (any type or color can be substituted)	160-1705-ND

Note that the parts, J1, U2, D1, C1 and C2 can be omitted if you have a 5V supply available. You can wire the supply in to the holes at the U2 position being careful of polarity.

Programming the PIC

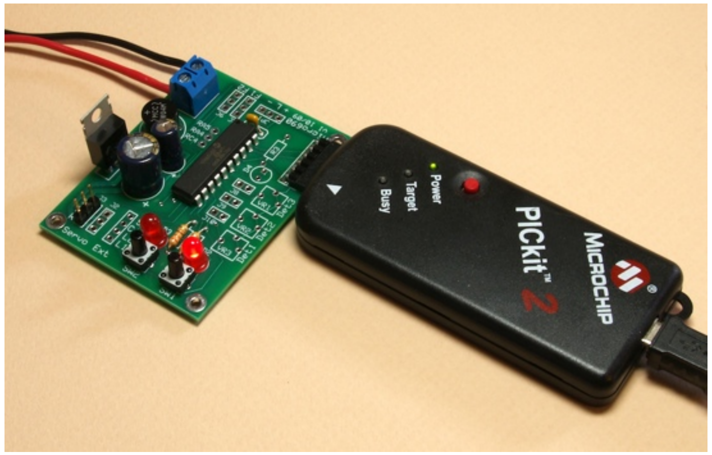


Fig. 7. The PICkit 2 programmer is being used to program the Micro690. The programmer is connected to J4. This can be done with or without the power connected.

The one piece of additional hardware needed is a PIC programmer (Fig. 7). This device connects to your computer and is used to download the program from the computer to the micro-controller's flash memory. On the Micro690, this is done through the header J4, the ICSP (In-Circiut Serial Programmer) header. There are a bewildering array of programmers available with all sorts of features. Microchip (www.microchip.com) makes a handy starter kit (Microchip Part No. DV164120, Digi-Key Part # DV164120-ND) which contains a PICkit 2 programmer with a software CD and a prototyping board with an included PIC16F690, which can also be used as a breadboard for building your own designs. You can also buy the programmer alone and save a few dollars (Digi-Key Part # PG164120-ND).

Alternatively, you can purchase pre-programmed chips at my site, Tam Valley Depot (www.tamvalleydepot.com/micro690). If you do this, you will not need a PIC programmer and you will not need the ICSP connector, J4. I will also be making a kit of parts available for building your own servo controller including the PC board.

Software

The software is by far the most complicated and difficult part of implementing a micro-controller. However, since I have done this part for you, it will make things much easier! I will explain the software in some detail so that you can understand it and also so you can modify it for your own design. I have written the software in C. While BASIC may be a more common choice for model railroad computer programs, there are downloadable, no-cost C compilers readily available for PIC chips. You can find other PIC code on the web but much of it is in assembly language, which is a real challenge to program in, difficult to understand and follow, and is not portable to other micro-controllers. Each family of micro-controllers has its own assembly language, whereas C is the same on all computers. I will make available pre-programmed chips, so if you do not wish to learn PIC programming, you can skip all the rest of this article!

An excellent book written specifically for programming the PIC16F690 using MPLAB and HiTech C is available. It uses the PICkit 2 started kit recommended above (which is why I picked it of course). See the A**dditional Resources** section at the end of the article. This book explains not only the PIC but also includes a C programming tutorial that will explain all those curly braces and semi-colons. In the sections below I explain various routines with some code fragments. You can find teh entire listing at the end of the chapter. You can also find a link to a downloadable file in the **Program Download** section below.

Servo Control

Controlling a servo requires a brief pulse every 20 milliseconds. The width of the pulse determines the position of the servo, 0.9 milliseconds is one end of the rotation and 2.1 milliseconds is the other end, with the middle of the rotation at 1.5 milliseconds. The heart of the program is an interrupt routine that utilizes two hardware timers on the PIC16F690 (almost all PIC chips have these same timers built in so this software can be considered a general PIC routine). The first timer, TMR0, times the 20 millisecond time between pulses and the second timer, TMR1, times the variable pulse width.

```
#define SwServo1  RA0          //servo output
unsigned int temp, sPos1;      // an int is 16-bits
bit isr_flag;                  // single bit flag variable
```

```
// Variables in caps are internal registers as defined in
// PIC16F690 documentation

//  The interrupt routine - all interrupts are routed here
void interrupt isr_routine(void)
{
      if (T0IF){                    // is this is a TMR0 interrupt?
            isr_flag = 1;           // set flag to let main routine
                                    // know interrupt occurred
            temp = 0xFFFF - sPos1;  // calculate TMR1 value
            TMR1ON = 0;             // turn off TMR1 while changing it
            TMR1L = temp&0x00ff;    // set TMR1 low bits
            TMR1H = temp>>8;        // set TMR1 high bits
            TMR1ON = 1;             // start TMR1
            SwServo1 = 1;           // turn on servo
            TMR0 = 256-155;         // reset TMR0
            T0IF = 0;               // clear interrupt flag
      } else {                //  otherwise it must be a timer 1 interrupt
            TMR1IF = 0;             //  clear the interrupt flag
            SwServo1 = 0;           // turn off servo
            move_servo();           // ramp the servo
      }
}
```

This interrupt routine is called every time an interrupt occurs - either from TIMR0 or from TIMR1 - so the first thing we do is check to see which timer caused the interrupt by checking the TMR0 interrupt flag, T0IF. If it is a TMR0 interrupt we turn the servo pulse on and then set TMR1 to interrupt after the pulse width time, according to the value in the variable `sPos1`, has elapsed. We then we reset TMR0 to interrupt again 20 milliseconds later. Otherwise, if it is a TMR1 interrupt (the else clause), we know we are at the end of a servo pulse, so we turn the servo off. We then call `move_servo()` to check to see if the servo needs to be ramped. This routine will be explained below. (PIC experts will note that TMR1 continues running, but since it is a 16-bit timer it will take much longer than 20 milliseconds to interrupt again and before that we will be setting it when TMR0 interrupts, so leaving it on does not matter.) The micro-controller will run this interrupt routine infinitely sending out a continuous stream of pulses to control the servo according the the value in `sPos1`.

In order to get the proper timing we need to set the proper values in several of the PIC16F690's control registers. First, we set the internal oscillator for 8 MHz with the OSCCON statement. The PIC16F690 divides the internal clock by 4 before sending it to the timers. This means that the clock period is 2 MHz or 0.5 microseconds per tick. To get 20 milliseconds we need to count 20000/0.5 or 40000 ticks. Since TMR0 is only 8-bits wide, the prescaler is used to further divide the clock to get it into range of 0-256. Setting the prescaler to 256 using the OPTION register gives us 40000/256 or 156.25. To allow for the overhead of the interrupt we will use 155. Since TMR0 counts up, not down, we actually load 256-155 into the TMRO register:

```
// setup code for timers and interrupts
OSCCON = 0b01110000;    // 8 MHz Internal Clock - bits <6:4> = 111
OPTION = 0b01000111;    // TMR0 prescaler divide by 256 - <2:0> = 111
TMR0 = 256-155;         // calculate TMR0 for 20 ms delay
```

TIMR1 is 16-bit so we don't need the prescaler. However, to make it easier to program it is convenient to use a prescaler of 2 so that we can use units of 1 microsecond ticks. Thus, a 2 millisecond pulse is thus 2000 ticks and a 1500 milliseconds pulse is 1500 ticks. Again, since TMR1 counts up, we will subtract this value from 65535 (0xFFFF in hexadecimal) and load this into the two registers for TMR1 TMR1L and TIMR1H after splitting the 2-byte into into the low and high portions. Now that the timers are ready to go we can enable the interrupts by turning on the TMR0 interrupt enable, T0IE, the timr1 interrupt enable, TMR1IE, the peripheral interrupt enable bit, PEIE and the global interrupt enable bit, GIE:

```
temp = 0xFFFF - sPos1;  // calculate TMR1 for servo pulse
TMR1H = temp>>8;        // roll temp right 8 bits to get the high byte
TMR1L = temp&0x00FF;    // mask out the high byte to get the low byte
T1CON = 0b00010001;     // TMR1 prescaler divide by 2 - bits <5:4> = 01
                        //    TMR1 on - bit <0> = 1
TMR1IE = 1;             // enable timer 1 interrupt
T0IE = 1;               // enable timer 0 interrupt
PEIE = 1;               // peripheral interrupts are enabled
GIE = 1;                // global interrupt enabled
```

Now that the interrupt routine is running, to change the position of the servo we just update the value in `sPos1` to the width of the desired pulse in microseconds and the interrupt routine will take care of the pulses needed in the background. We can now move on to writing the rest of the code to do just that.

```
// this subroutine ramps the servos up or down to make sPos1 = servo1Pulse
void move_servo(void){
      if (sPos1 < servo1Pulse ){
            sPos1+= servoSpeed;
            if (sPos1 > servo1Pulse) sPos1 = servo1Pulse;
      } else if (sPos1 > servo1Pulse ){
            sPos1-=servoSpeed;
            if (sPos1 < servo1Pulse) sPos1 = servo1Pulse;
      }
}
```

The subroutine `move_servo()` is used to ramp the servo gradually from one position to another. The value in `servo1Pulse` is the desired final pulse-width and `servoSpeed` controls the rate of change. Each time the routine is called, every 20 ms, it compares the value in `sPos1` with `servo1Pulse` and adds or subtracts the value in `servoSpeed` to move `sPos1` towards `servo1Pulse`. It does some checking to make sure `sPos1` doesn't overshoot. If the value of `sPos1` is the same as `servo1Pulse`, then it leaves `sPos1` unchanged. Note that the speed variable works inversely, that is, the higher the value, the faster the servo changes. A value of 1 causes the servo to move very slowly and a value of 100 moves about as fast as a servo can go. A value of 10 is about right for slow speed turnout movements.

```
// this subroutine sets the servos according to s1State
void set_servo(void)
{
      if ( s1State == Closed) {
            servo1Pulse = servoClosed;
            flashLED1 = 2;
      } else {
            servo1Pulse = servoThrown;
            flashLED2 = 2;
```

```
        }
        save_sys();         // remember the servo states
}
```

We use the routine `set_servo()` to set the value of `servo1Pulse` to the position we want. The servo is set to one of two values, `servoClosed` or `servoThrown` according to the value in `s1State`, the current servo state. Now all we have to do is change `s1State` from 0 to 1 and call `set_servo()` and the servo will start moving from one position to the next at the rate set by our speed value. The statement `save_sys()` remembers the the value of `s1State` in the on board EEPROM. It may not be needed in all programs. If for instance a toggle switch is used to control a turnout, the state of the toggle switch acts as a memory. If however, a pulse from a momentary switch is used to toggle the turnout, the routine is needed, as there is no outside source of the turnout state.

These three routines constitute a module for controlling a servo that can be used in a variety of programs. It can form the basis of numerous animation programs. I have left out the setup code needed at the beginning of the program to simplify the explanation. You will find it in the complete listing at the end of the article.

Lets move on to the `main()` routine of our program.

```
main()
      /*
            ...set up code goes here...
      */

      while(1){   // this loop repeats infinitely
            if (isr_flag) {              // wait for ISR flag
                  isr_flag = 0;
                  check_buttons();  // check for button changes
                  LED_handler();          // handle the leds
            }
      }
}
```

We create an infinite loop and within that loop we wait for the flag `isr_flag` to be set. Remember from the `isr_interrupt()` routine that this flag is set every time TMR0 goes off, that is, every 20 milliseconds. This makes a handy timing flag for the rest of the program. The loop resets the flag, checks first for any buttons pressed, and then updates the LEDs over and over ever 20 milliseconds. That's it. Well we have to write `check_buttons()` and `LED_handler()`. These are fairly complex routines in the final listing as we will see. The main reason for this is that as well as moving the servo the routines can be used to program the endpoint and the speed of the servo.

First, lets see some simple example routines that could be used to move the servo according to a toggle switch and to light a pair of LEDs to read out the current state.

```
#define Toggle1   RB0
#define LED1      RB1
#define LED2      RB2
void check_buttons(void)
{
      // Toggle1 is the pin with attached to a toggle switch
      if (Toggle1 != lastToggle1){
            s1State = Toggle1;
            set_servo();
            lastToggle1 = Toggle1;
      }
}

void LED_Handler()
{
      if ( s1State == 0 ) {
            LED1 = 1;
            LED2 = 0;
      } else {
            LED1 = 0;
            LED2 = 1;
      }
}
```

If the values for the servo endpoints and speed are known in advance, the values could be hardwired in to the program, and this would be all the code we need for a basic servo controller connected to a toggle switch.

This may seem like a complicated method for controlling a servo. It is possible to just directly program the pulse timing in an infinite loop in the main routine, and every example of servo control for a PIC that I found on the web and in books works this way. The difficulty arises when we want to add other tasks to the program, such as the LED handler and the button checking. Because these routines take various amounts of time which cannot be predicted on each cycle, sometimes our pulses will be delayed or come too soon. Even worse is if we need to run a calculation that would take much longer than 20 ms - say we want to time a delay of several seconds. By running the pulses off of the hardware timers in the background using the interrupt routine method, we eliminate these issues. If we were to do a very long calculation, the interrupts will keep the servo pulses coming regularly even while the PIC is otherwise engaged. The interrupt method lets the program have two essentially independent tasks running at the same time. This will be especially useful later in the series when we will be running routines that will take much longer than the 20 ms servo refresh rate.

Programming the Servo Endpoints

To be a really useful controller it would be nice to be able to program the endpoints and speed of the servo. To do this we will use the following convention. Initially the servo will start in the centered position to make it easier to install. We will use two pushbuttons and two LEDs to do the programming. If we hold one of the pushbuttons down for more than 0.5 seconds then the program will switch to training mode. The corresponding LED will start flashing to let the user know we are in training mode. In this mode brief pushes of the pushbuttons will move the servo endpoints. One pushbutton moves one way and the other pushbutton moves the opposite direction. When the other pushbutton is held we will change to the training the second endpoint and the second LED will start flashing to indicate this. To get out of programming we hold the flashing pushbutton down again and we will exit training mode. The endpoints values will be saved in the EEPROM memory. To set the speed we will hold down both pushbuttons at the same time. Both LEDs will flash to indicate speed training mode. In this mode, one pushbutton will speed the servo up and the other slow it down. So the user can see the speed change we will also toggle the servo position every time the pushbutton is pressed so the servo will move. When we are not in training mode the buttons and the LEDs will be used to control the servo.

We will use the single bit variables, `train1`, `train2`, and `trainSpeed` to flag the current training mode. When a pushbutton is pressed it will pull the pin it is attached to down to ground. Thus, a 1 indicates an open, unpressed pushbutton and a 0 indicates a pushbutton is being pressed. The tricky bit is determining if a pushbutton is pressed just briefly or if it is held down. To do that we need two counter variables, `but1Timer` and `but2Timer`, to count how long each pushbutton is pressed. Furthermore, a third variable per pushbutton, is needed to indicate that the pushbutton has been released and returned to a 1, `but1Armed` and `but2Armed`. Now, after the pushbutton has been pressed, we can use `but1Armed` to tell that pushbutton1 has been pressed and released.

```
// this subroutine handles button presses
// if not in training mode then a brief push of the buttons changes to that servo state
// long presses (> ~0.5 sec) puts us in training mode where the buttons now
// are used to adjust the servo endpoints
// finally, the sub checks to see if the remote switch input has changed
//  the remote switch allow for using more than one switch to change the turnout
//  useful on a module where a switch is needed on both sides
void check_buttons(void)
{
```

```
if( but1Timer == 50 ) {  // button held down for 0.02*50 = 0.5 secs
      if ( trainSpeed ) {  // already in training mode so end
            trainSpeed = train1 = train2 = 0;
      } else if ( but2Timer > 0 ) {
            // both buttons - speed training mode
            trainSpeed = 1;
            train1 = train2 = 0;
            but2Timer = 51;  // inhibit short press
      } else {
            // train servo endpoint 1 mode (train1)
            train1 = ~train1;  // enter opposite mode
            train2 = trainSpeed = 0;
            s1State = Closed;
            set_servo();
      }
      but1Timer = 51;
}
if( but1Armed == Armed ) {  // a brief button press
      if( train1 ==1) {   // move the closed endpoint up
            servoClosed = servoClosed + 5;
            servo1Pulse = servoClosed;
            save_sys();
      } else if (train2 == 1 ){  // move the thrown endpoint up
            servoThrown = servoThrown + 5;
            servo1Pulse = servoThrown;
            save_sys();
      } else if (trainSpeed == 1){  // increase the speed
            servoSpeed += 5;
            if( servoSpeed >= SpeedMax ) servoSpeed = SpeedMax;
            s1State = ~ s1State;    // change servo so user can see the change in speed
            set_servo();
      } else {    // not in training mode - change the servo state
            s1State = Closed;
            set_servo();
      }
      but1Armed = NotArmed;
}
```

```
// button2 - identical to button 1 for the most part
if( but2Timer == 50 ) {
      if ( trainSpeed ) {  // already in training mode so end
            trainSpeed = train1 = train2 = 0;
      } else if ( but1Timer > 0 ) {
            // both buttons - speed programming mode
            trainSpeed = 1;
            train1 = train2 = 0;
            but1Timer = 51;  // inhibit short presses
      } else {
            // train servo endpoint 2 mode (train2)
            train2 = ~train2;  // enter opposite mode
            train1 = trainSpeed = 0;
            s1State = Thrown;
            set_servo();
      }
      but2Timer = 51;
}
if( but2Armed == Armed ) {
      if( train1 ==1) {  // decrease the closed endpoint
            servoClosed = servoClosed -5;
            servo1Pulse = servoClosed;
            save_sys();
      } else if (train2 == 1 ){  // decrease the thrown endpoint
            servoThrown = servoThrown -5;
            servo1Pulse = servoThrown;
            save_sys();
      } else if (trainSpeed == 1){ // decrease the servo speed
            if (servoSpeed >= SpeedMin + 5 )servoSpeed -= 5;
            else servoSpeed = SpeedMin;
            s1State = ~ s1State;    // change servo so user can see the change in speed
            set_servo();
      } else {    // not in training mode - change the servo state
            s1State = Thrown;
            set_servo();
      }
      but2Armed = NotArmed;
```

```
		}
		// Check the state of the buttons
		if (Button1 == NotPressed){
			if(but1Timer > 0 ) {  // found falling edge since timer > 0
				if ( but1Timer < 50 ) {
					// if timer> 0 but less than 50 then this is a short press
					but1Armed = Armed;
				}
			}
			but1Timer = 0;   // reset timer
		} else {
			if ( but1Timer < 0xff )   // make sure we don't overflow the byte
				but1Timer++;
		}
		if (Button2 == NotPressed){
			if(but2Timer > 0 ) {  // found falling edge since timer > 0
				if ( but2Timer < 50 ) {
					// if timer> 0 but less than 50 then this is a short press
					but2Armed = Armed;
				}
			}
			but2Timer = 0;   // reset timer
		} else {
			if ( but2Timer < 0xff )   // make sure we don't overflow the byte
				but2Timer++;
		}
}
```

This just leaves LED_`handler()`. We will look at the values of `train1`, `train2` and `trainSpeed` and flash the LEDs accordingly. If we are not in training mode the LEDs will be used to indicate the servo state. To allow for flashing the LEDs in other parts of the program we use two variables `flashLED1` and `flashLED2` to control the LEDs. The variable is set to twice the number of flashes desired.

```
// This subroutine handles the LED according to the values in flashLED1/2
// also checks to see if we are in training or program mode and flashes
// leds accordingly
void LED_handler(void)
{
      LEDTimer--;
      if ( LEDTimer == 0){
            if( trainSpeed ) {            // if training speed flash both LEDs
                  if( flashLED1 == 0 ) flashLED1 = 2;
                  if( flashLED2 == 0 ) flashLED2 = 2;
            } else if ( train1 ) {  // if training endpoint1 flash LED1
                  if( flashLED1 == 0 ) flashLED1 = 2;
                  LED2 =0;
            } else if ( train2 ) {  // if training endpoint2 flash LED2
                  if( flashLED2 == 0 ) flashLED2 = 2;
                  LED1 = 0;
            } else {          // otherwise set the LED to sState if not flashing an LED
                  if ( flashLED1 == 0 ) LED1 = s1State;
                  if ( flashLED2 == 0 ) LED2 = ~s1State;
            }
            if ( flashLED1 > 0 ) {
                        // if flashLED is even light the LED
                        if ((flashLED1 & 0x01) == 0) {
                              LED1 = 1;
                        } else {    // if it is odd turn it off
                              LED1 = 0;
                        }
                        flashLED1--;      // subtract 1
            }
            if ( flashLED2 > 0 ) {
                        if ((flashLED2 & 0x01) == 0) {
                              LED2 = 1;
                        } else {
                              LED2 = 0;
                        }
                        flashLED2--;
            }
            LEDTimer = 10;    // wait 10 more interrupt cycles = 0.2 seconds
      }
}
```

EEPROM Routines

Finally we need to implement `save_sys()` and `restore_sys()`. These two routines store the servo variables within the EEPROM so that the values will be saved when the power gets turned off. A flag, `do_restore`, is used by `restore_sys()` to reset all the values to the factory defaults. At power up, we will check to see if either one of the pushbuttons is pressed and, if it is, we will set this flag to force a factory value reset.

```
      // reset memory to defaults if either button pressed on power up
      if( Button1 == Pressed ) {
            delay(5);                   // delay 5 milliseconds
            if (Button1 == Pressed ){   // read the button again
                  doRestore = 1;        // if still pressed do the restore
                  but1Timer = 51;       // inhibit the button from changing
                                        // the servo state
            }
      } else if(Button2 == Pressed ) {
            delay(5);
            if (Button2 == Pressed ){
                  doRestore = 1;
                  but2Timer = 51;
            }
      }

      restore_sys();              // restore the values from memory
```

Note that we don't just use the first value read but wait to see if the button is still pressed. This prevents accidental noise spikes (not unlikely just after a power-up) from erasing our memory.

Most of the work in `restore_sys()` and `save_sys()` are done in the library routines `eeprom_read()` and `eeprom_write()` which come with the C compiler. The only tricky bit is to make sure you write and read the variable from the same memory address!

```
// Reads saved values from EEPROM memory for
// servo throw endpoints, servo speed and servo state
// If doRestore is set upon entry, or if the memory
// has never been written to, then it sets these values to
// the factory defaults.
void restore_sys (void)
{
      if (doRestore) {
restore:
            // restore to factory defaults
            doRestore = 0;
            version = VersionNo;
            s1State = Closed;
            servoThrown = ServoCenter - 30;  // small movement so we can
            servoClosed = ServoCenter + 30;  // tell its alive
            servo1Pulse = servoClosed;
            servoSpeed = 10;
            save_sys();       // save the new values
            flashLED1 = 4;    // lots of flashes so we know it happened
            flashLED2 = 4;
            return;
      } else {
            // version is an arbitrary number to mark the memory as previously written to
            version =   eeprom_read( 0x00 );
            if( version != VersionNo ) goto restore;  // blank EEPROM - restore from
defaults
            servoThrown =     eeprom_read( 0x03 );
            servoThrown +=    (Word)(eeprom_read( 0x04 )<<8);
            servoClosed =     eeprom_read( 0x05 );
            servoClosed +=    (Word)(eeprom_read( 0x06 )<<8);
            servo1Pulse =     eeprom_read( 0x07 );
            servo1Pulse +=    (Word)(eeprom_read( 0x08 )<<8);
            s1State =   (bit)eeprom_read( 0x09 );
            servoSpeed  = eeprom_read( 0x0A );
      }
}
```

```
// Writes the servo values to EEPROM
// Make sure that the memory locations are in sync between
// restore_sys() and save_sys()
// Plenty of room left in the 256-byte EEPROM (PIC16F690)
// for saving other variables
void save_sys(void){
      eeprom_write(0x00, version);
      eeprom_write(0x03, (Byte)(servoThrown&0x00FF));
      eeprom_write(0x04, (Byte)(servoThrown>>8));
      eeprom_write(0x05, (Byte)(servoClosed&0x00FF));
      eeprom_write(0x06, (Byte)(servoClosed>>8));
      eeprom_write(0x07, (Byte)(servo1Pulse&0x00FF));
      eeprom_write(0x08, (Byte)(servo1Pulse>>8));
      eeprom_write(0x09, (Byte)s1State);
      eeprom_write(0x0A, (Byte)servoSpeed);
}
```

That is all the routines we need to implement our servo driver and controller. You can use the servo driver as described here for an animation project or for throwing turnouts. In the next article we will add the flashing lights, in the third article we will add optical train detection and in the final article I will describe how to install the Micro690 on your layout.

Program Download

The program listing can be downloaded from the TamValleyRR website. To compile it requires HITech C PICC LITE and the MPLAB IDE. This code will work on the PIC16F677/685/687/689/690 family of chips with no modifications and other PIC16F chips that have EEPROM, TIMR0 and TIMR1 running at an 8 MHz clock rate - others may require modifications. Note the configuration bits comments in the code - these will need to be set in MPLAB for the chip to work correctly!

Additional Resources

Books

“Beginner's Guide to Embedded C Programming” by Chuck Hellebuyck, published by Electronic Products, 2008. Highly recommended if you are new to C and programming PICs. Uses the PICKit 2 and the PIC16F690 also used in this article.

“Programming 8-bit Microcontrollers in C with Interactive Hardware Simulation” by Martin P. Bates, published by Newnes, 2008. Covers a wide variety of topics for a number of different PIC16 chips including many types of peripherals and advanced techniques. An excellent book for those familiar with the basics of PIC chips.

Suppliers

www.digikey.com - My favorite source of electronic parts
www.mouser.com - Another good source of electronic parts
www.tamvalleydepot.com - Micro690 PC boards, preprogrammed PIC16F690s and a kit of parts. (Disclaimer: the author owns this site.)

Information on PICs

www.microchip.com - Download and additional information for PIC16F690, MPLAB and the PICkit 2 programmer.
www.htsoft.com - Download and additional information for HItech C PICC LITE.

Complete Program Listing

```
//
//
#include <htc.h>
typedef unsigned char Byte;
typedef unsigned int Word;


// pins
#define SwServo1  RC5   // Servo output

#define LED1            RB4   // LED output
#define Button1         RB5 // Button input
#define LED2            RB6   // LED output
#define Button2         RB7   // Button input
#define ExtSwitch RA2   // Ext Switch input

//  Configuration bits (set in MPLAB/Cofigure/Configuration Bits)
//          OSC         Internal RC No Clock
//          WDT         Off
//          PUT         Off
//          MCLRE Internal
//          CP          Off
//          CPD   off
//          BODEN BOD and SBOREN disabled
//          IESO  Enabled
//          FCMEN Enabled

//  Constants
//
#define     VersionNo   0x28

// servo
#define     ServoCenter 1500        // = 1.500 ms
#define     ServoMax    900   // = .9 ms
#define     ServoMin    2100  // = 2.1 ms
```

```
#define     Pressed     0
#define     NotPressed  1
#define     Armed       1
#define     NotArmed    0
#define     Closed      1
#define     Thrown      0
#define     SwitchOn    1
#define     SwitchOff   0
#define     SpeedMax    100
#define     SpeedMin    1

bit isr_flag;
bit doRestore;
bit servosOn;
bit but1Armed;
bit but2Armed;
bit train1;
bit train2;
bit trainSpeed;
bit s1State;
bit lastSwitch;

Byte version;
Word servoThrown;
Word servoClosed;
Word servo1Pulse = ServoCenter;
Byte servoSpeed = 10;

Word temp;

Word sPos1 = ServoCenter;
Byte but1Timer;
Byte but2Timer;
Byte moveTimer;
Byte flashLED1;
Byte flashLED2;
Byte LEDTimer = 1;
Byte switchTimer = 1;

//   Subroutine declarations
void move_servo(void);
void set_servo(void);
void save_sys(void);
void restore_sys(void);
void delay( Word t );

void check_buttons(void);
void set_servos(void);
void check_short(void);
void LED_handler(void);
void DCC_receive(void);

void interrupt isr_routine(void)
{
      if (T0IF){              // is this is a timer0 interrupt?
            isr_flag = 1;           // set flag to let main routine know intterrupt occured
            temp = 0xFFFF - sPos1;  // calculate timer1 value
            TMR1ON = 0;             // turn off timer1 while changing it
            TMR1L = temp&0x00ff;    // set timer1 low bits
            TMR1H = temp>>8;        // set timer1 high bits
            TMR1ON = 1;
            SwServo1 = 1;           // turn on servo
            TMR0 = 256-155;         // reset timer0
            T0IF = 0;               // clear intterrupt flag
      } else {                //  otherwise it must be a timer 1 intterrupt
            TMR1IF = 0;             //  clear the interrupt flag
            SwServo1 = 0;           // turn off servo
            move_servo();           // ramp the servo
      }
}

main()
{
```

```
// Setup starts here
      OSCCON = 0b01110000;    // 8Mhz Internal Clock
      PORTA = 0;              // clear PortA
      PORTB = 0;              // clear PortB
      PORTC = 0;              // clear PortC
      ANSEL = 0;              // turn off analog channels
      ANSELH = 0;             // turn off analog channels
      TRISA = 0b11111111;     // set RA0 to an output
      WPUA = 0b0000100;       // enable weak pullups on PORTA
      TRISB = 0b10101111;     // set port B I/O directions
      WPUB = 0b10100000;      // enable weak pullups on PORTB
      TRISC = 0b11011111;     // set port C I/O directions
      OPTION = 0b01000111;    // TMR0 prescaler set to 256
      temp = 0xFFFF - sPos1;  // calculate timer1 for servo pulse
      TMR1H = temp>>8;
      TMR1L = temp&0x00FF;
      T1CON = 0b00010001;     // TMR1 prescaler = 2, TMR1 on
      TMR0 = 256-155;         // calculate timer0 for 20 ms delay
      TMR1IE = 1;             // enable timer 1 interrupt
      T0IE = 1;               // enable timer 0 interrupt
      PEIE = 1;               // peripheral interrupts are enabled
      GIE = 1;                // global interrupt enabled
      T0IF = 1;               // force timer0 interrupt right away to start servo
      flashLED1 = 2;          // flash LEDs
      flashLED2 = 2;
      // reset memory to defaults if either button pressed on power up
      if( Button1 == Pressed ) {
            delay(5);
            if (Button1 == Pressed ){     // Check twice to avoid noise spikes
                  doRestore = 1;
                  but1Timer = 51;
            }
      } else if(Button2 == Pressed ) {    // Check the other button
            delay(5);
            if (Button2 == Pressed ){     // Check twice to avoid noise spikes
                  doRestore = 1;
                  but2Timer = 51;
            }
      }
      restore_sys();          // restore the values from memory
      // init the servos to stored values
      if( s1State == Closed ) servo1Pulse = servoClosed;
      else servo1Pulse = servoThrown;
      sPos1 = servo1Pulse;
      lastSwitch = ExtSwitch;       // external switch setup
      // end of setup

      // Wait for an interrupt then poll inputs - forever...
      while(1){   // this loop repeats infinitely
            if (isr_flag) {               // wait for ISR flag
                  isr_flag = 0;
                  check_buttons();  // check for button changes
                  LED_handler();          // handle the leds
                  //move_servo();         // ramp the servo a bit
            }
      }
}

// This subroutine ramps the servos up or down to make sPos1 = servoPulse
void move_servo(void){
      if (sPos1 < servo1Pulse ){
            sPos1+= servoSpeed;
            if (sPos1 > servo1Pulse) sPos1 = servo1Pulse;
      } else if (sPos1 > servo1Pulse ){
            sPos1-=servoSpeed;
            if (sPos1 < servo1Pulse) sPos1 = servo1Pulse;
      }
}

// This subroutine sets the servos according to s1State
void set_servo(void)
{
      if ( s1State == Closed) {
            servo1Pulse = servoClosed;
            flashLED1 = 2;
      } else {
            servo1Pulse = servoThrown;
            flashLED2 = 2;
      }
      save_sys();       // remember the servo states
}


//this subroutine handles the LED according to the values in flashLED1/2
// also checks to see if we are in training or program mode and flahes leds accordingly
void LED_handler(void)
{
      LEDTimer--;
      if ( LEDTimer == 0){
            if( trainSpeed ) {            // if training speed flash both LEDs
                  if( flashLED1 == 0 ) flashLED1 = 2;
                  if( flashLED2 == 0 ) flashLED2 = 2;
            } else if ( train1 ) {  // if training endpoint1 flash LED1
                  if( flashLED1 == 0 ) flashLED1 = 2;
                  LED2 =0;
            } else if ( train2 ) {  // if training endpoint2 flash LED2
                  if( flashLED2 == 0 ) flashLED2 = 2;
                  LED1 = 0;
            } else {          // otherwise set the LED to sState if not flashing an LED
                  if ( flashLED1 == 0 ) LED1 = s1State;
                  if ( flashLED2 == 0 ) LED2 = ~s1State;
            }
            if ( flashLED1 > 0 ) {
                        // if flashLED is even light the LED
                        if ((flashLED1 & 0x01) == 0) {
                              LED1 = 1;
                        } else {    // if it is odd turn it off
                              LED1 = 0;
                        }
                        flashLED1--;      // subtract 1
            }
            if ( flashLED2 > 0 ) {
                        if ((flashLED2 & 0x01) == 0) {
                              LED2 = 1;
                        } else {
                              LED2 = 0;
                        }
                        flashLED2--;
            }
            LEDTimer = 10;    // wait 10 more interrupt cycles = 0.2 seconds
      }
}

// this subroutine handles button presses
// if not in training mode then a brief push of the buttons changes to that servo state
// long presses (> ~0.5 sec) puts us in training mode where the buttons now
// are used to adjust the servo endpoints
// finally, the sub checks to see if the remote switch input has changed
//  the remote switch allos for using more than one switch to change the turnout
//  useful on a module where a switch is needed on both sides
void check_buttons(void)
{
      if( but1Timer == 50 ) {
            if ( trainSpeed ) {  // already in program mode so end
                  trainSpeed = train1 = train2 = 0;
            } else if ( but2Timer > 0 ) {
                  // both buttons - DCC programming mode
                  trainSpeed = 1;
                  train1 = train2 = 0;
                  but2Timer = 51;  // inhibit short press
            } else {
                  // train servo endpoit 1 mode (train1)
                  train1 = ~train1;  // enter opposite mode
                  train2 = trainSpeed = 0;
                  s1State = Closed;
                  set_servo();
            }
            but1Timer = 51;         // prevents interpreting as short press
      }
      if( but1Armed == Armed ) { // short press
            if( train1 ==1) {       // increase servoClosed
                  servoClosed = servoClosed + 5;
                  servo1Pulse = servoClosed;
                  save_sys();
            } else if (train2 == 1 ){ // increase servoThrown
                  servoThrown = servoThrown + 5;
                  servo1Pulse = servoThrown;
                  save_sys();
            } else if (trainSpeed == 1){  // increase speed
                  servoSpeed += 5;
                  if( servoSpeed >= SpeedMax ) servoSpeed = SpeedMax;
                  s1State = ~ s1State;    // change servo so user can see the change in speed
                  set_servo();
            } else {
                  s1State = Closed;
                  set_servo();
            }
            but1Armed = NotArmed;
      }
      // button2
      if( but2Timer == 50 ) {
            if ( trainSpeed ) {  // already in program mode so end
                  trainSpeed = train1 = train2 = 0;
            } else if ( but1Timer > 0 ) {
                  // both buttons - DCC programming mode
                  trainSpeed = 1;
                  train1 = train2 = 0;
                  but1Timer = 51;  // inhibit short press
            } else {
                  // train servo endpoit 2 mode (train2)
                  train2 = ~train2;  // enter opposite mode
                  train1 = trainSpeed = 0;
                  s1State = Thrown;
                  set_servo();
            }
```

```
			but2Timer = 51;
		}
		if( but2Armed == Armed ) {
			if( train1 ==1) {	// decrease servoClosed
				servoClosed = servoClosed -5;
				servo1Pulse = servoClosed;
				save_sys();
			} else if (train2 == 1 ){ // decrease servoThrown
				servoThrown = servoThrown -5;
				servo1Pulse = servoThrown;
				save_sys();
			} else if (trainSpeed == 1){  // decrease speed
				if (servoSpeed >= SpeedMin + 5 )servoSpeed -= 5;
				else servoSpeed = SpeedMin;
				s1State = ~ s1State;    // change servo so user can see the change in speed
				set_servo();
			} else {
				s1State = Thrown;
				set_servo();
			}
			but2Armed = NotArmed;
		}
		// re-arm buttons
		if (Button1 == NotPressed){
			if(but1Timer > 0 ) {  // found falling edge since timer > 0
				if ( but1Timer < 50 ) {
					// if timer> 0 but less than 50 then this is a short press
					but1Armed = Armed;
				}
			}
			but1Timer = 0;   // reset timer
		} else {
			if ( but1Timer < 0xff ) but1Timer++;
		}
		if (Button2 == NotPressed){
			if(but2Timer > 0 ) {  // found falling edge since timer > 0
				if ( but2Timer < 50 ) {
```

```
                        // if timer> 0 but less than 50 then this is a short press
                        but2Armed = Armed;
                  }
            }
            but2Timer = 0;    // reset timer
      } else {
            if ( but2Timer < 0xff ) but2Timer++;
      }
      //  check the remote switch
      // change state if edge detected outside of 25*.20 second limit
      if ( switchTimer > 0 ) {
            switchTimer--;
      }
      if ( switchTimer == 0 ) {
            if ( ExtSwitch != lastSwitch ) {
                  s1State = ~s1State;
                  set_servo();
                  switchTimer = 25;
            }
      }
      lastSwitch = ExtSwitch;
}

// delay x milliseconds (approximate!)
// especially if an interrupt occurs in the middle
Byte delcntr;
void delay(Word t)
{
      Word i;
      for(i=0; i<t*2; i++){
#asm
            clrf _delcntr
delay1      nop
            ;nop
            incfsz _delcntr
            goto delay1
#endasm
```

```
      }
}

// Reads saved values from EEPROM memeory for
// servo throw endpoints, servo speed and servo state
// If doRestore is set uon entry, or if the memory
// has never been written to, then it sets these values to
// the factory defaults.
void restore_sys (void)
{
      if (doRestore) {
restore:
            // restore to factory defaults
            doRestore = 0;
            version = VersionNo;
            s1State = Closed;
            servoThrown = ServoCenter - 30;  // small movement so we can tell its alive
            servoClosed = ServoCenter + 30;
            servo1Pulse = servoClosed;
            servoSpeed = 10;
            save_sys();       // save the new values
            flashLED1 = 4;    // lots of flashses so we know it happened
            flashLED2 = 4;
            return;
      } else {
            // version is an arbitrary number to mark th memory as previously written to
            version =   eeprom_read( 0x00 );
            if( version != VersionNo ) goto restore;  // blank EEPROM - restore from
defaults
            servoThrown =     eeprom_read( 0x03 );
            servoThrown +=    (Word)(eeprom_read( 0x04 )<<8);
            servoClosed =     eeprom_read( 0x05 );
            servoClosed +=    (Word)(eeprom_read( 0x06 )<<8);
            servo1Pulse =     eeprom_read( 0x07 );
            servo1Pulse +=    (Word)(eeprom_read( 0x08 )<<8);
            s1State =   (bit)eeprom_read( 0x09 );
            servoSpeed  = eeprom_read( 0x0A );
      }
}

// Writes the servo values to EEPROM
// Make sure that the memory laocatsion are in synce between
// restore_sy() and save_sys()
// Plenty of room left in the 256-byte EEPROM (PIC16F690)
// for saving other variables
void save_sys(void){
      eeprom_write(0x00, version);
      eeprom_write(0x03, (Byte)(servoThrown&0x00FF));
      eeprom_write(0x04, (Byte)(servoThrown>>8));
      eeprom_write(0x05, (Byte)(servoClosed&0x00FF));
      eeprom_write(0x06, (Byte)(servoClosed>>8));
      eeprom_write(0x07, (Byte)(servo1Pulse&0x00FF));
      eeprom_write(0x08, (Byte)(servo1Pulse>>8));
      eeprom_write(0x09,      (Byte)s1State);
      eeprom_write(0x0A,      servoSpeed);
}
```