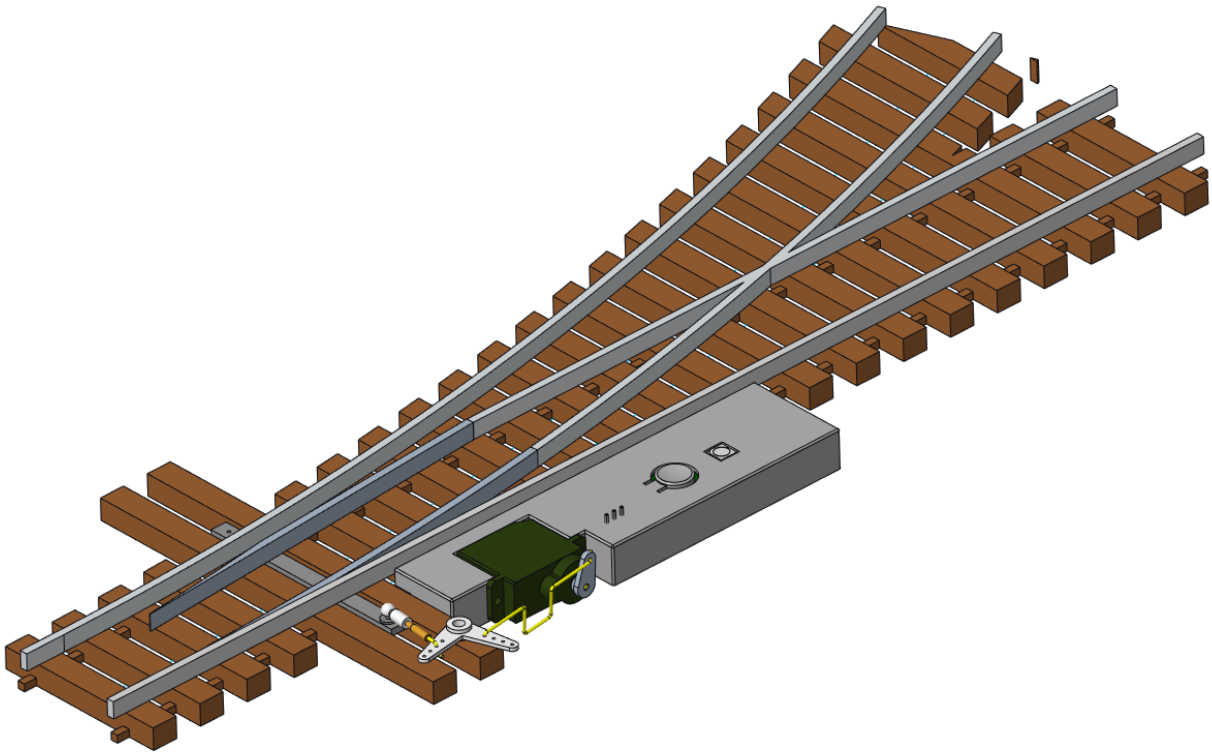

An Arduino-based DCC Accessory Decoder for Model Railroad Turnouts

Eric Thorstenson

11/1/17

Introduction

Earlier this year, I decided to develop an Arduino-based DCC accessory decoder for model railroad turnouts. I wanted the completed decoder/turnout assembly to satisfy a few key design objectives:

- The turnout must be easily installed on the layout with no external wiring.
- The controller must be small enough to be installed 'above board' on the turnouts.
- It should be software upgradeable and programmable in place.

In addition, I wanted the design to support several variants – a basic design for the majority of turnouts, a version with relays for longer turnouts, and a version for crossover control.

What I ended up with turned out to be a fun but substantial project…

Design Overview

The project involved the development of the electrical and software elements of the decoder itself, as well as its mechanical and electrical interface to the turnout. The completed assembly includes a custom built printed circuit board providing the electrical interface for the Arduino, a 3D printed enclosure, and the mechanical linkages to drive the turnout. The assembled turnout with the decoder on it is both powered and controlled by the DCC signal from the track, and requires no additional wiring.

The completed design has the following features:

- Powered and controlled by DCC directly from the track, with no additional wiring - completely plug and play
- Operation configurable via CVs
- Software upgradeable in place
- Controllable via accessory commands for normal turnout operation
- Controllable via signal aspect commands for external outputs and optional functions
- Non-derail sensors (contact or optical) automatically throw the turnout for trains approaching from the wrong direction
- Two solid state relays for powering or grounding switch rails as needed
- RGB LED for status indication
- Manual control via pushbutton on enclosure
- 1.5A switching power supply powers the board, servos, and externally accessible 5V output
- Two external controllable outputs for lighting or other accessories
- Supports up to four servos for complete crossover control
- Servos are powered off when not in use
- Two servo speeds, slow for normal operation and fast for non-derail

Figure 1 shows a block diagram of the overall system.

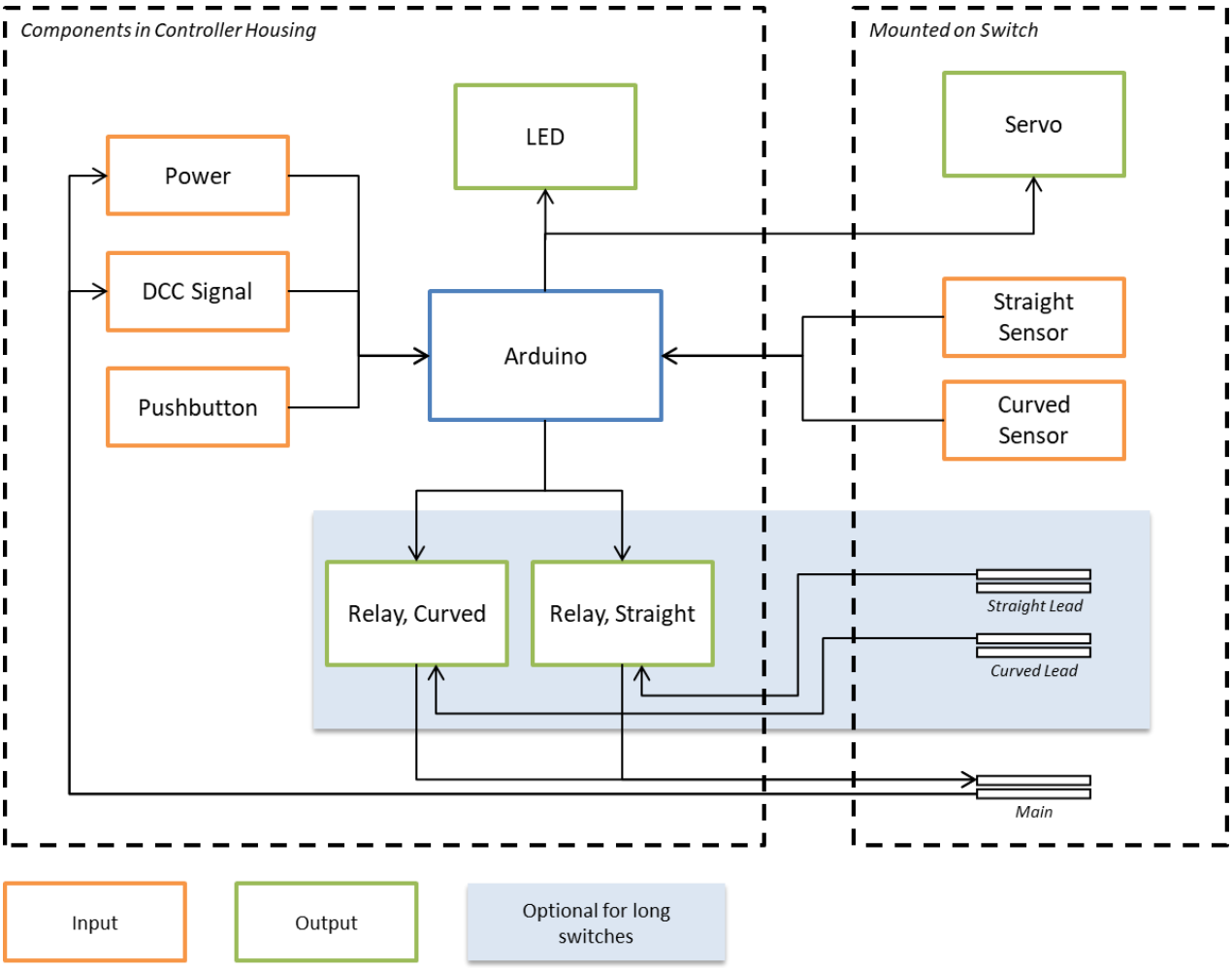

Figure 1 System Block Diagram

The electronics are contained in a 3D printed enclosure that mounts to the straight side of the turnout. The enclosure also houses the servo, which is connected to the points via a linkage and bellcrank mechanism. Power and the contact sensors are connected via wires soldered to the underside of the turnout rails. A pushbutton on the backside of the printed circuit board provides manual control of the turnout.

A standard hobby servo is used to throw the points. Position and speed settings are configurable using CVs. By default, it uses a 2.5 second motion for normal operation, and a 0.25 second motion when triggered by an occupancy sensor. The servo is powered off after the motion is completed, so it does not make noise or consume power when not in use.

The occupancy sensors for the non-derail feature can be configured either as contact closure sensors (operating in a manner similar to the traditional Lionel non-derail feature) or as optical infrared sensors, or both. The RGB LED provides position indication, as well as feedback for motion and programming. The solid state relays can be configured to provide power or ground to switch rails as needed for improved power delivery to engines over long switches.

The microprocessor controlling the whole unit is an Arduino Mini Pro running at 5V and 16MHz. It handles the receipt and decoding of the DCC commands, monitors sensors, and controls the various outputs. Power and the DCC signal are obtained from the rails of the switch.

Mechanical Design

The mechanical design consists primarily of the servo and actuator mechanism, and the electronics enclosure. Due to the nature of my layouts (large, 'temporary', above board), the mechanism and enclosure must both be small and low profile. Figure 2 shows the CAD model of the controller and mechanism installed on a Ross O72 turnout. Figure 14 shows a photo of the completed assembly.

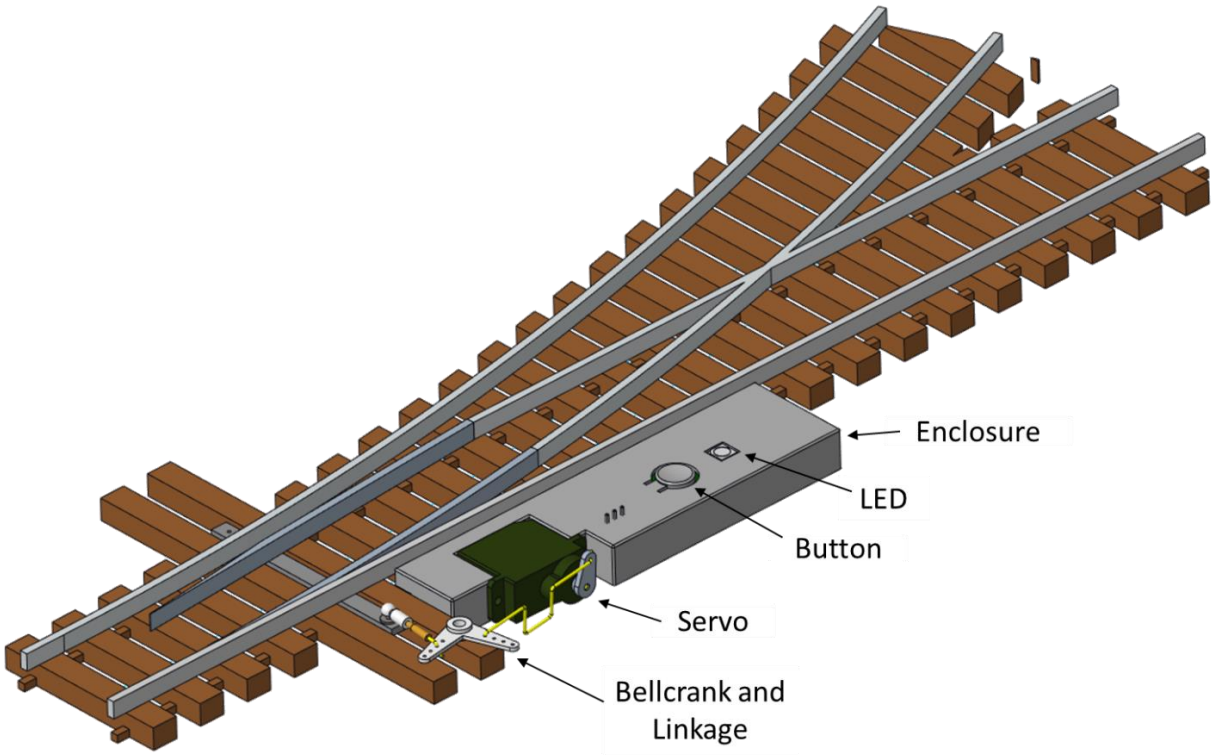

Figure 2 Turnout and Controller

Mechanism

Actuation for driving the points comes from a standard hobby servo controlled by the Arduino. A linkage, bellcrank, and ball and socket assembly transfers the servo's rotational motion to a linear motion applied laterally across the turnout. The servo endpoints are adjustable via CVs to allow positioning of the points. A dog-leg in the linkage provides compliance during normal operation, allowing the point preload to be maintained consistently. It also prevents damage to the turnout in case of servo malfunction, or in case a train is run through the wrong way while the controller is unpowered.

With the exception of the servo horn, the entire mechanism fits below the envelope of the top rail. The servo horn protrudes slightly above that, but remains outside the NMRA clearance template.

Figure 3 shows the model of the mechanism in the straight and curved positions. Figure 15 shows a close-up photo of the completed mechanism.

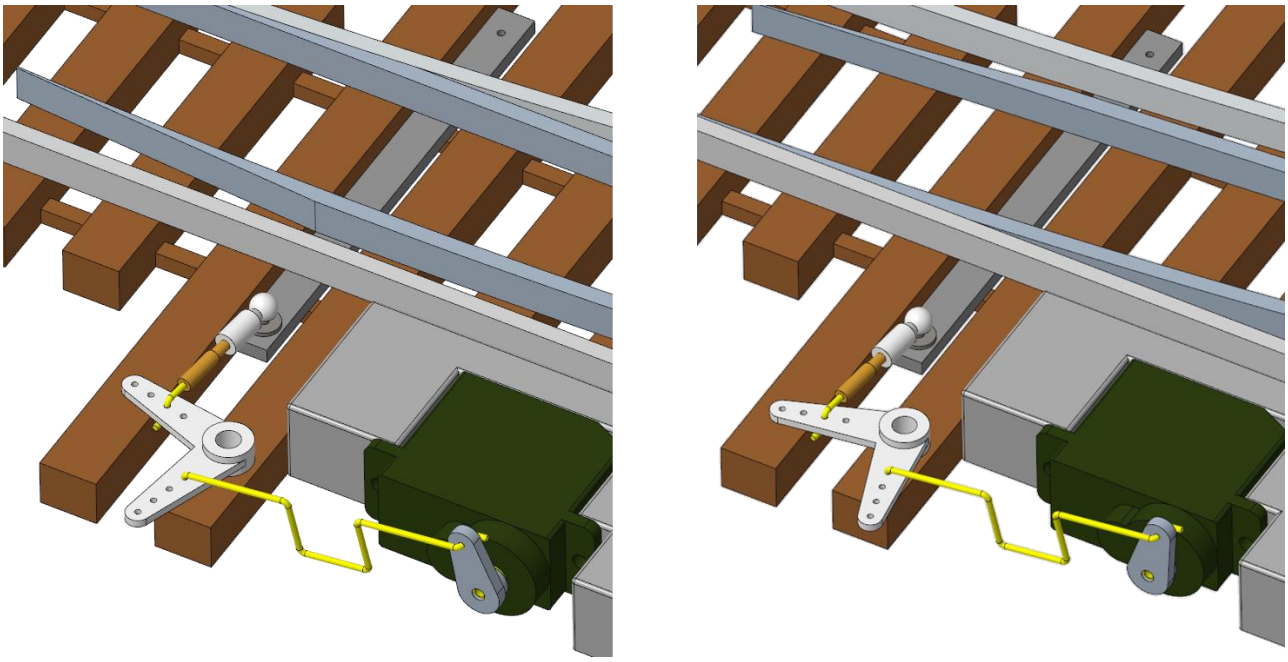

Figure 3 Mechanism, Straight and Curved Positions

Enclosure

The enclosure houses the PCB and forms the mounting interface for the servo. It also provides features for the button, LED, and pin header access. The enclosure is 3D printed from sintered nylon, providing the flexibility for the button to operate the tactile switch on the PCB. The enclosure is designed to be bonded to the ties of the turnout. The servo and PCB are then mounted to the enclosure using screws. Left and right hand versions of the enclosures are required corresponding to the left and right hand turnouts. The entire enclosure assembly fits below the envelope of the top rail.

Figure 4 shows views of the top and bottom of the standard enclosure, used for most turnouts. Figure 5 shows the enclosure for long switches and the enclosure for the crossover controller. Figure 16 shows a photo of the standard enclosure. Figure 17 shows the underside of the enclosure with the controller installed.

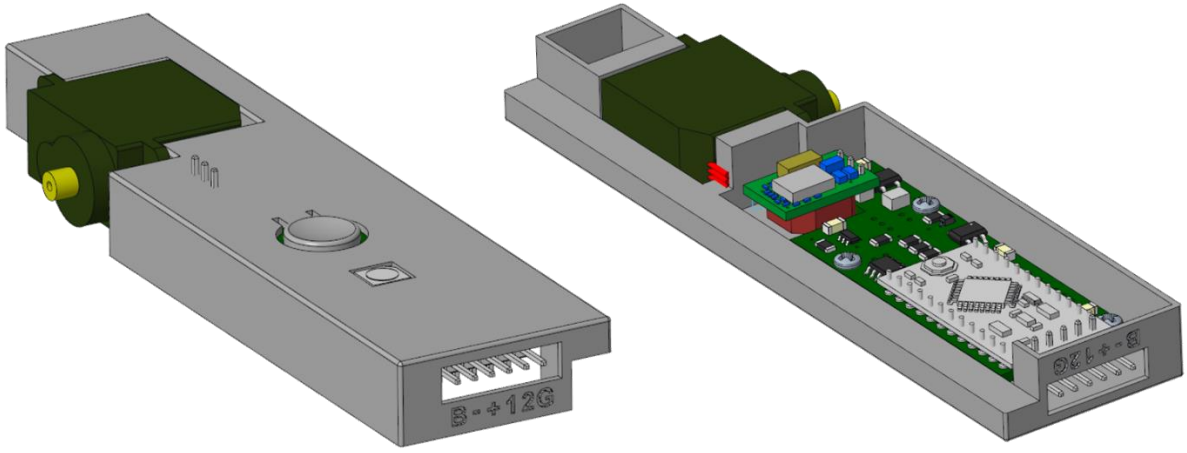

Figure 4 Standard Enclosure, Top and Bottom Views

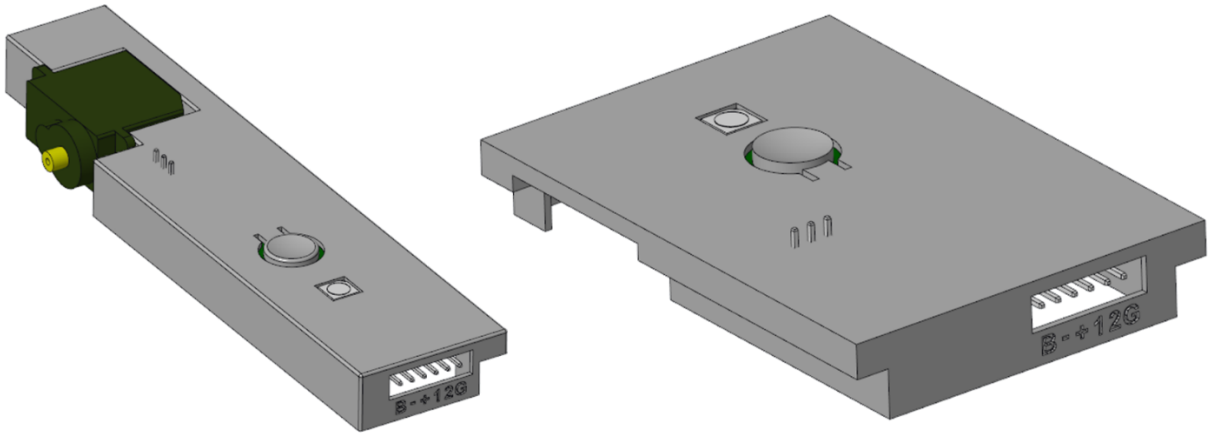

Figure 5 Long Enclosure and Crossover Enclosure

Electrical Design

The electrical design consists mainly of the power section, the DCC interface, the servo control, non-derail sensors, and the relays.

Power

Power for the decoder is provided by a switching regulator module powered by rectified track DCC voltage. The power module provides 5VDC at up to 1.5A to the Arduino, sensors, and optional external accessories. A common mode noise filter reduces noise from the power booster or operating engines. A 10uF capacitor smooths the output of the rectifier for input to the regulator.

DCC Interface

The DCC signal is provided to the Arduino using a high speed active output optocoupler. Track DCC current in the forward LED direction is detected by the opto, and causes the output to go low. Current in the reverse direction travels through the bypass diode and causes the output to go high. The opto uses a totem pole output, providing a sharply defined square wave that is fed directly to the input pins on the Arduino. Pins for both the hardware interrupt and the input capture register receive this signal, so the software can be configured to use either.

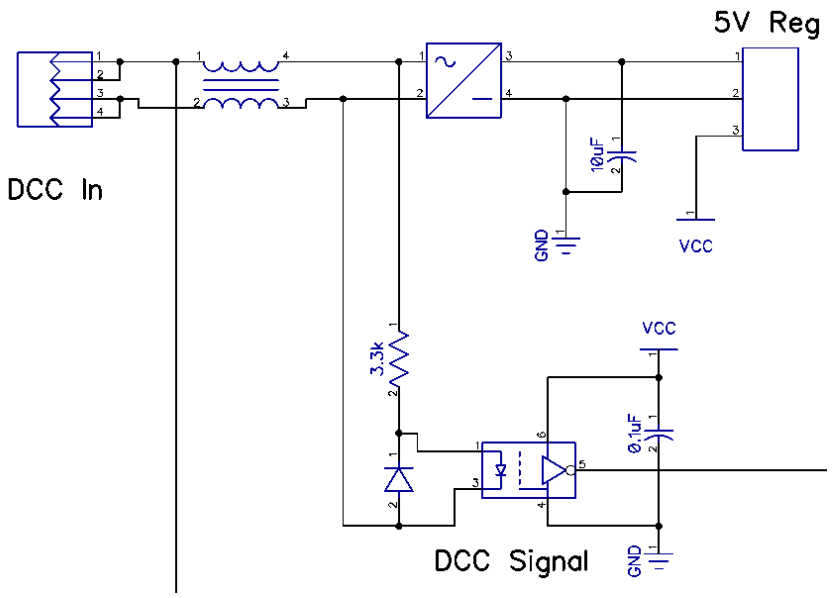

Figure 6 DCC Power and Signal

Servo Control

Although the Arduino PWM output can control the servo without any additional circuitry, I also wanted to power off the servo when it wasn't in use, to improve efficiency and eliminate noise. To do this, I used a high side power switch controlled by a separate output from the Arduino. Servo power is enabled after starting the PWM signal for the servo, and is disabled shortly after the servo motion is completed. The power switch is current-limited to protect the controller against excessive current draw.

In the crossover controller, all four servos are powered from a common power switch.

Non-Derail/Occupancy Sensors

A key feature that I wanted in the design was automatic non-derailing, so that trains approaching from the 'wrong' side of the turnout will trigger the controller to throw the turnout for them. I also wanted to detect occupancy so that the turnout cannot be thrown against a train sitting on it.

The design supports two ways of doing this. The first uses a contact closure approach, much like the old Lionel turnouts. An AC optocoupler detects when a 'positive' sense rail is 'grounded' by an axle contacting the sense rail and the outside turnout rail. Output from the opto provides an indication to the controller that that leg of the turnout is occupied.

An optional approach uses IR detectors with logic level outputs placed between the ties of each leg of the turnout. The output from these sensors is in parallel with the output from the optocouplers, and goes to the same inputs on the Arduino.

The controller can be installed on the turnout with either the contact closure sensors or the IR sensors. I use the contact closures on most switches because it is cheaper and easier to install that way. I use the IR sensors on longer turnouts (for example, Ross #6s) where I want all the ground rails to carry current. In either case, the controller logic is the same. The controller will set the turnout to accommodate an approaching train detected by a sensor. The controller will also ignore DCC or button commands that would cause the turnout to be thrown against an occupied leg of the turnout.

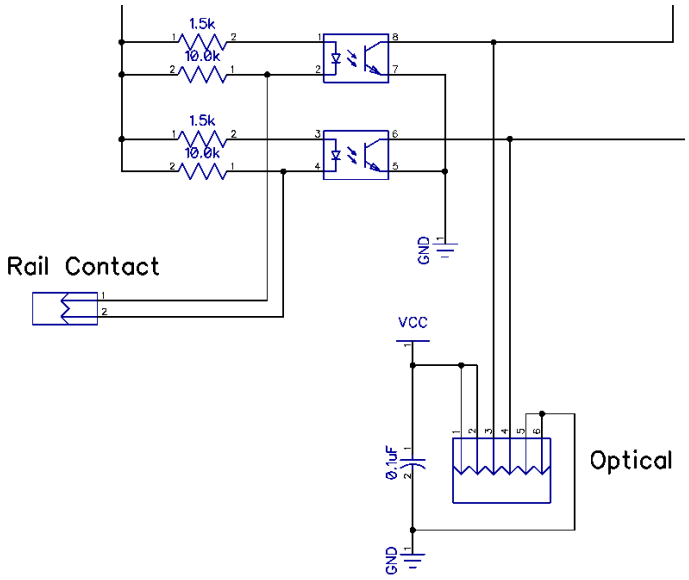

Figure 7 Non-derail Sensors

Relay Control

A variant of the controller design supports multiple relay outputs for powering or grounding track segments in response to changes in the turnout position. Outputs from the Arduino drive high current solid-state relays through current limiting resistors. The outputs of the relays are configured so that they can be used to power or ground rails, depending on the installation. For example, in the case of the long Ross turnouts, the relays are used to ground the straight and curved leads of the turnout. The relays are disabled before any servo motion begins, and are enabled at the conclusion of the servo motion.

PCB Layout

The basic controller PCB is a 0.9” x 2.5” two layer design. All the surface mount components except for the button and the RGB LED are on the top side of the board, so that it can be assembled in a single reflow operation. The button, LED, power module, and the Arduino itself are installed separately. A six pin header is installed on the Arduino for programming and for power and accessory operation once the turnout is installed.

The PCB for the long turnout version accommodates two relays, and measures 0.9” x 3.0”. The PCB for the crossover controller has four relays and is 1.5” x 2.4”.

Figure 18 and Figure 19 show photos of the completed PCB.

Software Design

The software design consists of three main components – the DCC decoding, the hardware input/output, and the overall management of the turnout. The following sections provide an overview of each of the classes and describe their interaction. Detailed descriptions and example usages are provided in the header comments for each class.

The completed software required development of twelve new class libraries and approximately 3500 lines of code.

DCC Decoding

The DCC decoding functionality is broken into three classes, each with a specific function. The BitStream class handles the capture of the raw DCC bitstream. It captures the raw pulse transitions in a queue, performs low level error checking to identify valid bits, and assembles and provides the captured bitstream via a callback. The Arduino input capture register is used in order to get the most accurate timing of the pulses, and to eliminate the influence of other ISRs that may be running. Bitstream capture using a hardware interrupt is also supported. Appendix A summarizes tests done comparing the various timer and interrupt options for the bitstream capture.

The DCCpacket class takes the raw bitstream and assembles it into valid DCC packets. It optionally enforces the DCC checksum, and can filter repeated DCC packets so that upstream classes can treat the packet delivery as reliable. Completed packets are provided to the DCCdecoder class via callback.

The DCCdecoder class takes a complete DCC packet and processes it, determining the packet type, parsing out the DCC address, and extracting the packet data. Callbacks for the various packet types provide the data to upstream classes.

The only part of the DCC decoding process that happens in an ISR is adding the pulse timer count to the queue – all other bitstream processing and packet decoding takes place as a normal process outside the ISR, significantly easing the constraints on processing time for the packet building, decoding, and other functions. The packet building and decoding may in fact be interrupted by the bitstream ISR if they run long, without the risk of missing a bit or degrading the packet processing.

The bitstream class is the only class that requires an actual DCC signal and an Arduino to unit test. The DCCpacket and DCCdecoder classes can be unit tested in any C environment, simplifying the use of test cases to verify performance.

Hardware Input/Output

Apart from the DCC input discussed above, the primary hardware I/O required for operation of the decoder includes a button input, an RGB LED output, one or more servo outputs, and occupancy sensor inputs. Relay outputs are provided for powering or grounding rails on the turnout where needed. In addition, two external auxiliary outputs are provided for low current loads such as LED lighting.

The Button class provides a basic debounced button input. It is used for the button on the enclosure, as well as for the occupancy sensor inputs, which may be either optical or contact closure inputs.

The OutputPin class is a simple wrapper around the digitalWrite functions for an output pin. It is used for the relays and two auxiliary outputs.

The RGB LED class facilitates control of the three output pins for the LED, as well as providing ON, OFF, and FLASH functionality.

The TurnoutServo class manages the control of the servo that drives the points of the turnout. It allows setting of the servo endpoints as well as high and low rate speeds. The servo motion is controlled based on the endpoints and the desired speeds. A callback provides notification of completed motion. Multiple servos may be controlled by the turnout manager for use on a crossover/turnout assembly.

Turnout Management

The overall management of the turnout is handled by the TurnoutMgr class. It provides the top level logic for acting on received DCC commands, responding to the occupancy sensors and button inputs, and controlling the servo, relays, and LED. It maintains the configuration of the turnout via CVs stored to

EEPROM, allows changes via DCC program on main commands, and provides a reset-to-default option. It handles updating any objects that require time-based updates (servo, button, LED), and provides error handling and indication in the case of repeated DCC bitstream or packet errors. Two auxiliary outputs as well as temporary configuration settings are controllable using extended accessory (signal aspect) commands.

The servo power pin is turned on and off as needed, so that the servo is only powered when it is actually in use. The PWM signal is started before the power is enabled and stopped after the power is disabled, so that the servo always has a valid signal while it is powered. In the case of the crossover manager, the servo motions happen sequentially, followed by turning off the servo power pin.

A derived class provides management for a crossover, controlling four servos and four relays.

Appendix A: Effect of Timer Configuration on Bitstream Capture

To keep the ISR code used in the bitstream capture as simple and quick as possible, I implemented a simple queue, whereby the ISR just puts a timestamp in the queue, and then non-ISR code can deal with decoding it. This makes it easy to change how I obtain the timestamp, as well as allowing considerably more flexibility in the code that interprets the timestamps.

As part of the development of the bitstream capture libraries, I examined the effect of different timer and interrupt configurations on the accuracy of the recorded timings. To assess the accuracy of the various options, I captured 250 samples for each method and plotted the calculated pulse width as a function of time (all times in microseconds). The tests were done using a Sparkfun Redboard with the ATmega328 running at 16MHz. The red and green lines in each plot indicate the limits for 1 and 0 bits, respectively.

Figure 8 shows the pulse widths calculated using a hardware interrupt and the micros() call to get the timestamp. The 4µs resolution of the micros() call is evident.

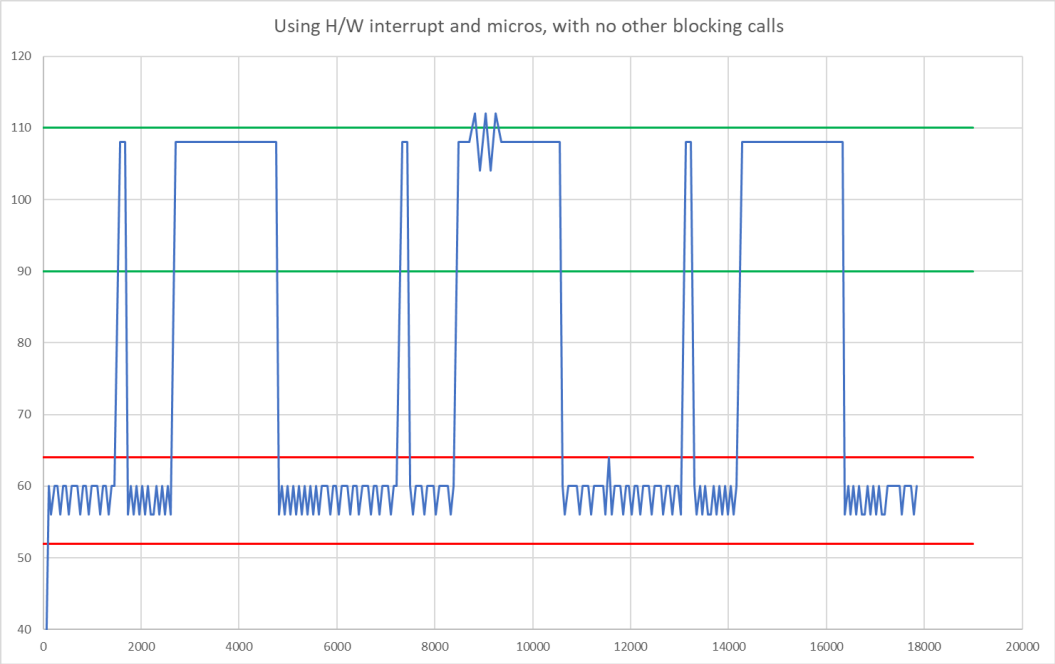

Figure 8 H/W Interrupt and micros()

Figure 9 shows the pulse widths using a hardware interrupt with the timestamp taken from the timer1 counter TCNT1. For this case timer1 was configured with a prescaler of 8, which gives 0.5µs resolution, with an overflow every 32ms.

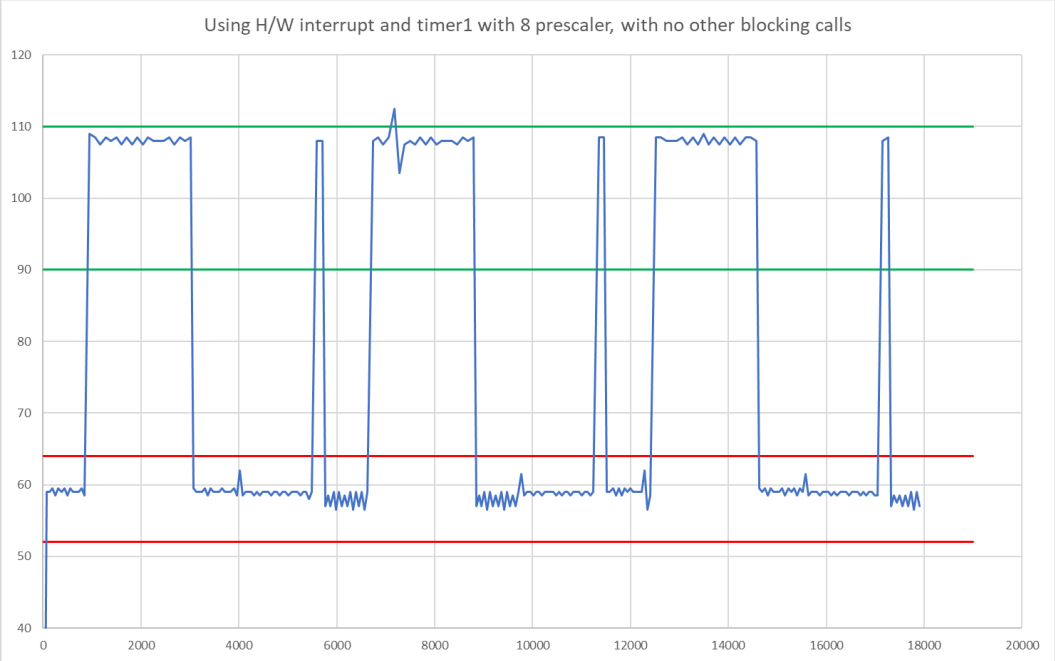

Figure 9 H/W Interrupt and Timer1 with 8 Prescaler

Figure 10 shows the effect on the timestamps of a 10µs noInterrupts block in main(), to simulate the servicing of other interrupts. Since the hardware interrupt only gets serviced outside that block, the recorded timestamps lose a similar degree of accuracy. This effect is evident in a couple of spots in Figure 8 and Figure 9 as well.

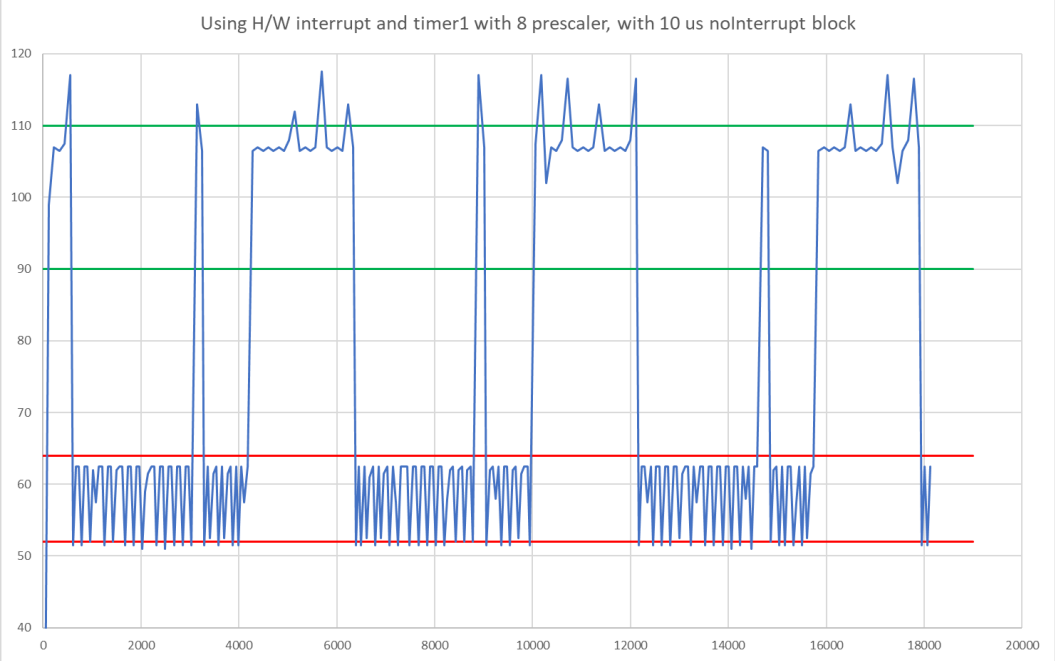

Figure 10 H/W Interrupt with 10µs noInterrupts Block

Figure 11 and Figure 12 show the pulse widths based on a timestamp from the input capture register. As above, timer1 was configured with a prescaler of 8, which gives 0.5µs resolution. Because the timestamp is captured in the register independent of when the ISR gets called, a delay in calling the ISR does not affect the accuracy of the timestamp. This is shown in Figure 12, with the same 10µs noInterrupts block as in Figure 10.

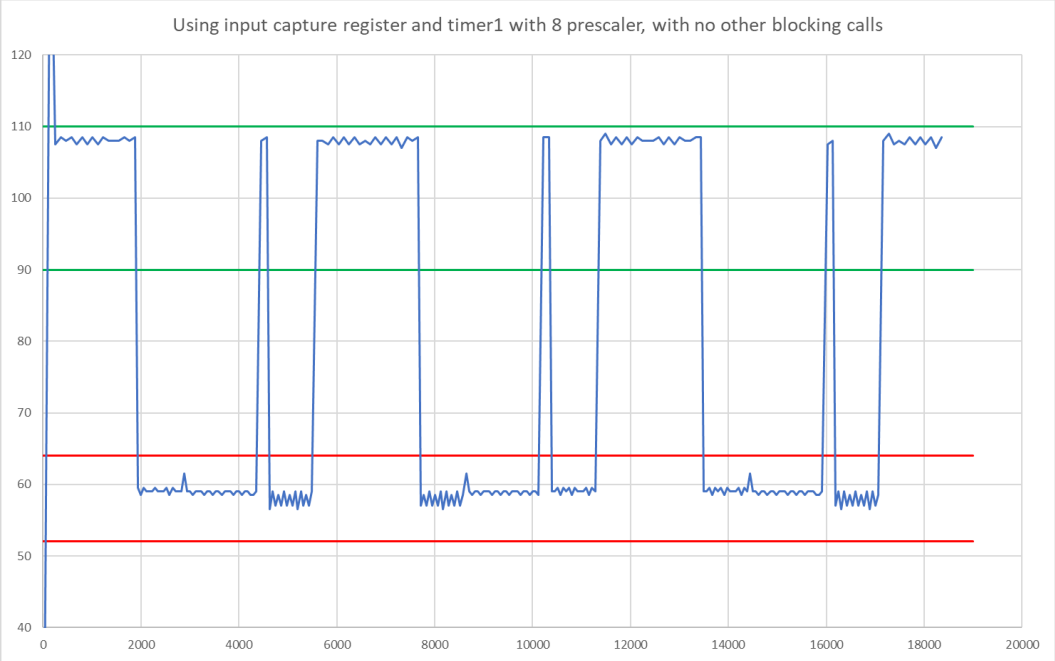

Figure 11 Input Capture Register

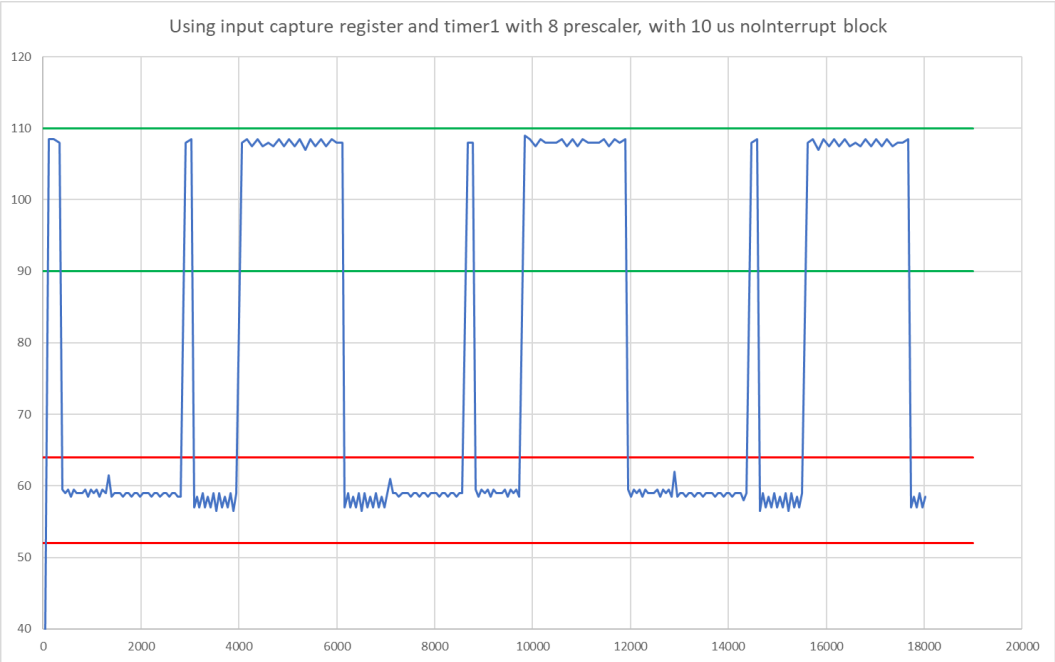

Figure 12 Input Capture Register with 10µs noInterrupts Block

Finally, Figure 13 shows the pulse widths based on a timestamp from the input capture register, with timer1 configured with no prescaler. This gives 0.0625µs resolution, with an overflow every 4ms. Although the 0.5µs resolution in Figure 11 and Figure 12 stays reliably within the 1 and 0 limits, the curve really takes shape here.

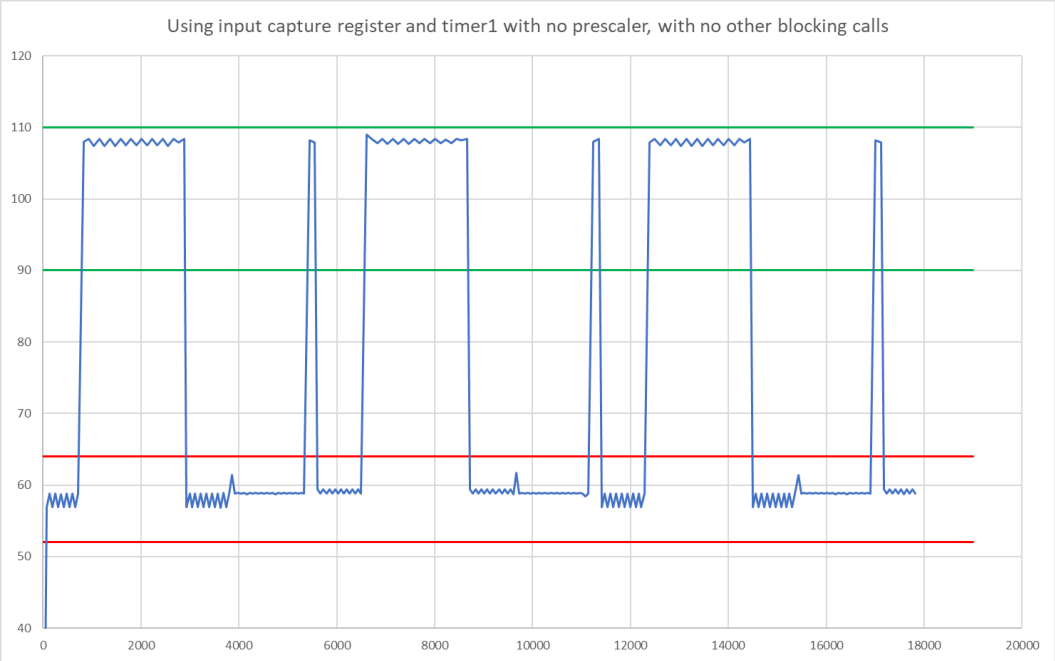

Figure 13 Input Capture Register with no Prescaler

Appendix B: Photos

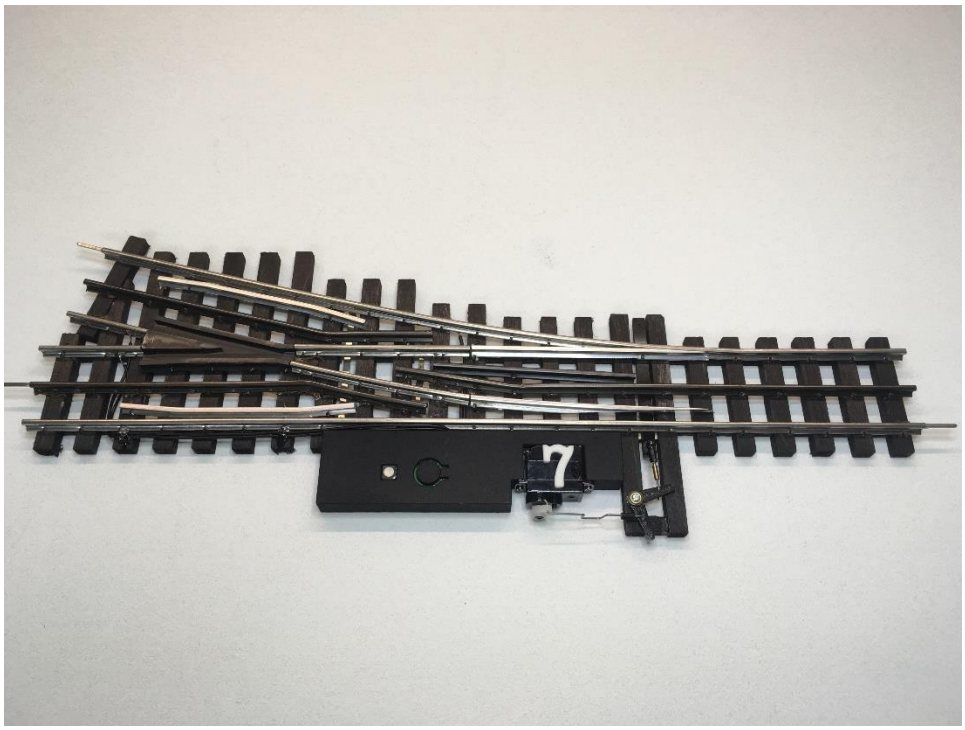

Figure 14 Completed Turnout

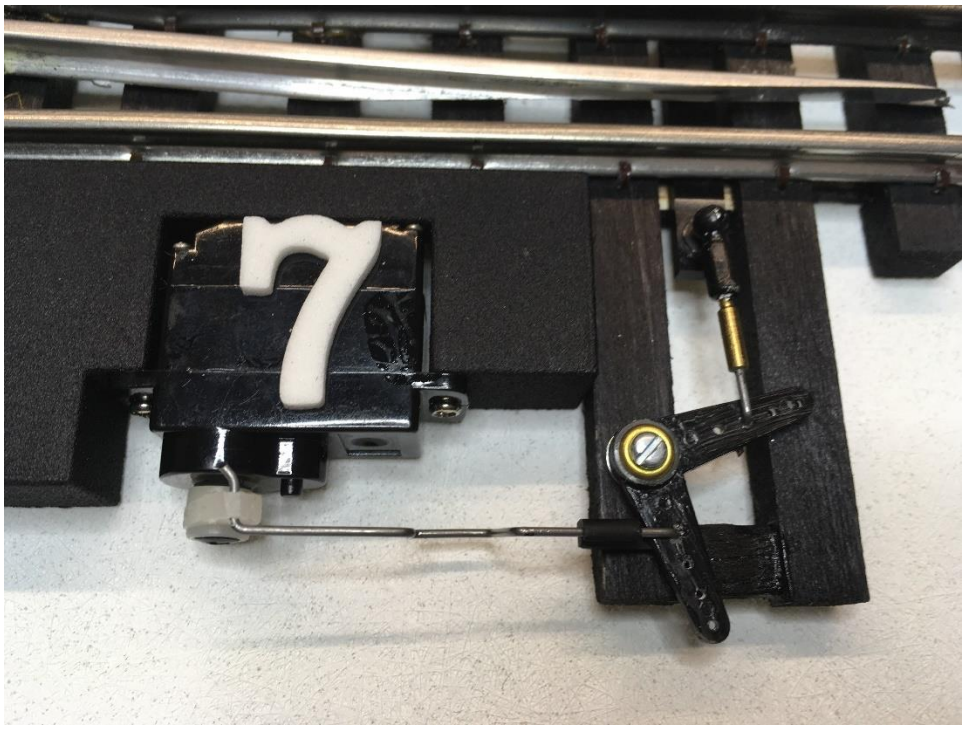

Figure 15 Servo and Mechanism

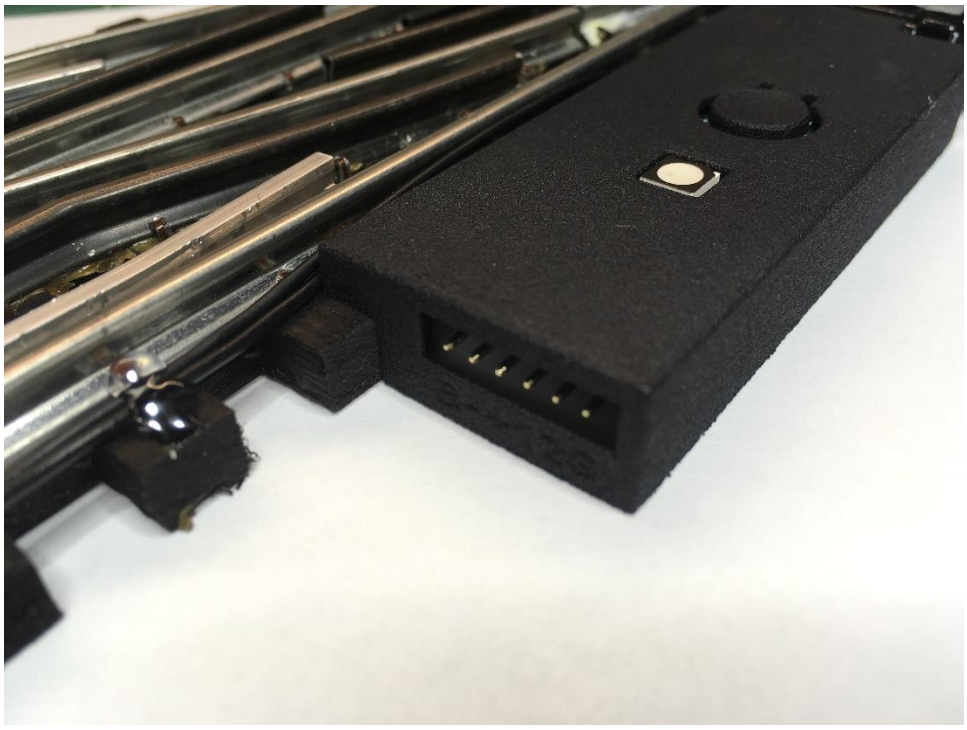

Figure 16 LED, Button, and Pin Header

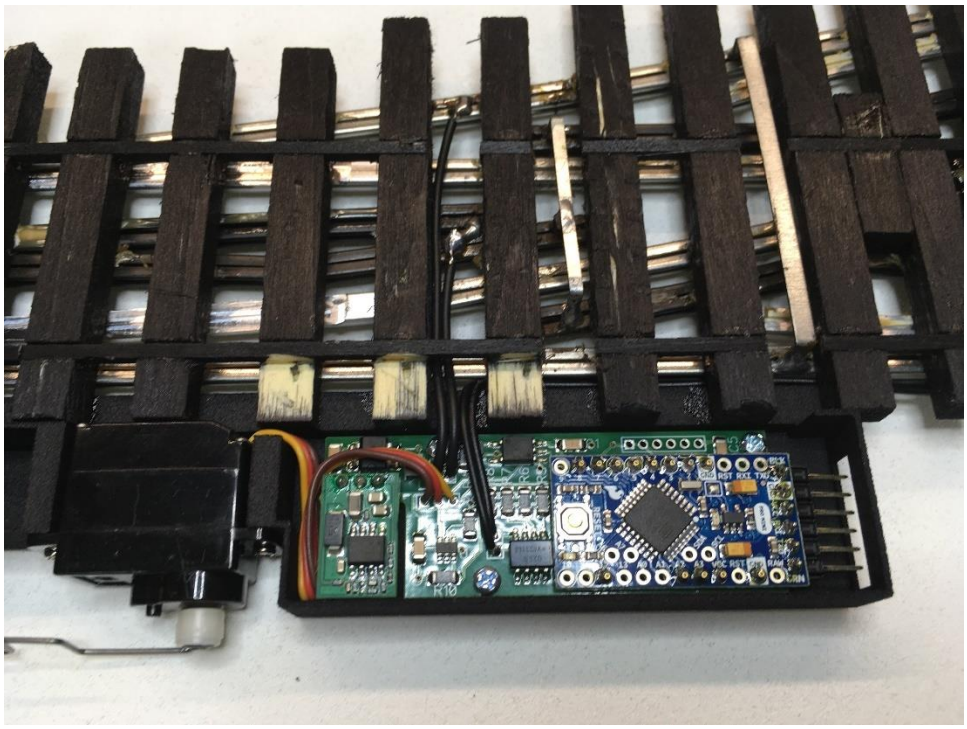

Figure 17 Underside of Installed Controller

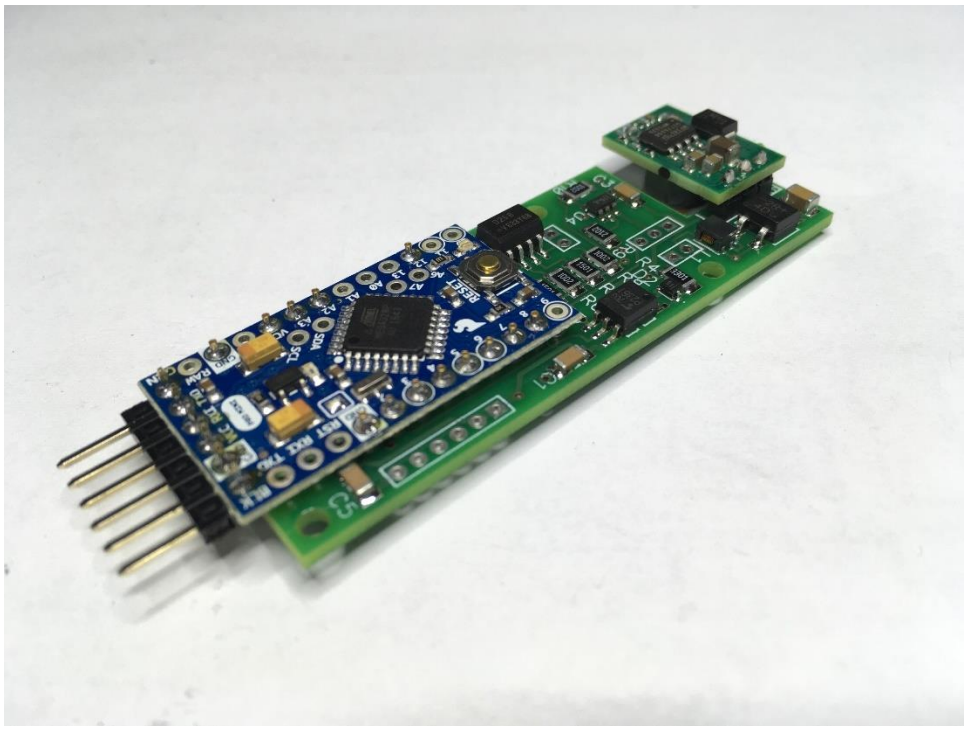

Figure 18 PCB Underside

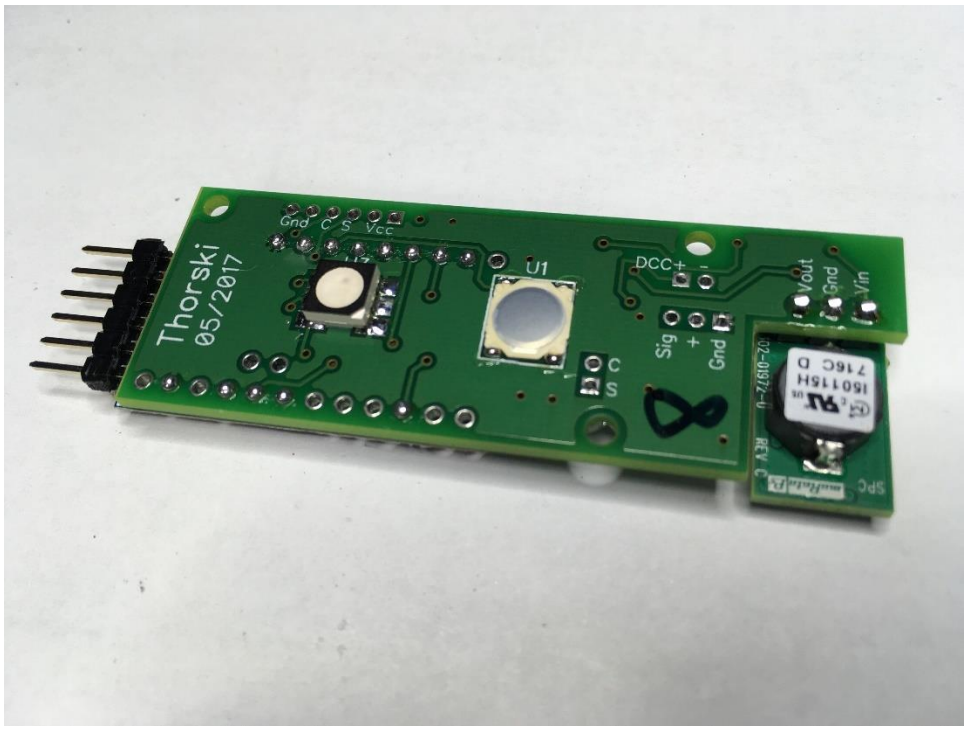

Figure 19 PCB Top Side