```perl
# =============================================================================
# FILE: DnB_Turnout.pm                                               8/12/2020
#
# SERVICES:  DnB TURNOUT FUNCTIONS
#
# DESCRIPTION:
#    This perl module provides turnout related functions used by the DnB model
#    railroad control program.
#
# PERL VERSION: 5.24.1
#
# =============================================================================
use strict;
# -----------------------------------------------------------------------------
# Package Declaration
# -----------------------------------------------------------------------------
package DnB_Turnout;
require Exporter;
our @ISA = qw(Exporter);

our @EXPORT = qw(
   I2C_InitServoDriver
   ProcessTurnoutFile
   InitTurnouts
   MoveTurnout
   SetTurnoutPosition
   GetTemperature
   TestServoAdjust
   TestTurnouts
);

use DnB_Message;
use Forks::Super;
use POSIX 'WNOHANG';
use Time::HiRes qw(sleep);

# =============================================================================
# FUNCTION:  I2C_InitServoDriver
#
# DESCRIPTION:
#    This routine initializes the turnout servo I2C driver boards on the DnB
#    model railroad. It sets parameters that are common to all servo ports. The
#    Adafruit 16 Channel Servo Driver utilizes the PCA9685 chip. The pre_scale
#    calculation is from the PCA9685 documentation.
#
#    Initialization sequence.
#       1. Get current ModeReg1.
#       2. Put PCA9685 into sleep mode.
#       3. Set servo refresh rate.
#       4. Normal mode + register auto increment.
#       5. Put PCA9685 into normal mode.
#
# CALLING SYNTAX:
#    $result = &I2C_InitServoDriver($BoardNmbr, $I2C_Address);
#
# ARGUMENTS:
#    $BoardNmbr      Drive board number being initialized.
#    $I2C_Address    I2C Address
#
# RETURNED VALUES:
```

```
#    0 = Success,  1 = Error.
#
# ACCESSED GLOBAL VARIABLES:
#    None.
# =============================================================================
sub I2C_InitServoDriver {

   my($BoardNmbr, $I2C_Address) = @_;
   my($result, $driver, $mode_data);

   my($minAddr, $maxAddr) = (0x40, 0x7F);  # AdaFruit 16 Channel PWM board range.
   my(%PCA9685) = ('ModeReg1' => 0x00, 'ModeReg2' => 0x01, 'AllLedOffH' => 0xFD,
                   'PreScale' => 0xFE);
   my($normal_mode) = 0xEF;   my($sleep_mode) = 0x10;   my($auto_inc) = 0xA1;

   my($freq) = 105;   # Refresh rate; 105 = 300-900 SG90 min/max position.

   my($pre_scale) = int((25000000.0 / (4096 * $freq)) - 1);

   &DisplayDebug(2, "I2C_InitServoDriver, BoardNmbr: $BoardNmbr   " .
                    "I2C_Address: $I2C_Address   pre_scale: $pre_scale");

# Validate that address is within the Adafruit 16-channel driver range.
   if ($I2C_Address >= $minAddr and $I2C_Address <= $maxAddr) {
      $driver = RPi::I2C->new($I2C_Address);
      unless ($driver->check_device($I2C_Address)) {
         &DisplayError("I2C_InitServoDriver, Failed to initialize " .
                       "I2C address: " . sprintf("0x%.2x",$I2C_Address));
         return 1;
      }
      $driver->write_byte(0x10, $PCA9685{'AllLedOffH'});  # Orderly shutdown.
      sleep 0.01;                                # Wait for channels to stop.
      $mode_data = $driver->read_byte($PCA9685{'ModeReg1'});
      $driver->write_byte(($mode_data | $sleep_mode), $PCA9685{'ModeReg1'});
      $driver->write_byte($pre_scale, $PCA9685{'PreScale'});
      $mode_data = ($mode_data & $normal_mode) | $auto_inc;
      $driver->write_byte(($mode_data), $PCA9685{'ModeReg1'});
      &DisplayDebug(2, "I2C_InitServoDriver, PreScale: " .
                       $driver->read_byte($PCA9685{'PreScale'}));
      undef($driver);
   }
   else {
      &DisplayError("I2C_InitServoDriver, Invalid I2C address: " .
                    "$I2C_Address   Board: $BoardNmbr");
      return 1;
   }
   return 0;
}

# =============================================================================
# FUNCTION:  ProcessTurnoutFile
#
# DESCRIPTION:
#    This routine reads or writes the specified turnout data file. Used to
#    retain turnout operational data between program starts.
#
# CALLING SYNTAX:
#    $result = &ProcessTurnoutFile($FileName, $Function, \%TurnoutData);
#
# ARGUMENTS:
```

```perl
#    $FileName       File to Read/Write
#    $Function       "Read" or "Write"
#    $TurnoutData    Pointer to %TurnoutData hash.
#
# RETURNED VALUES:
#    0 = Success,  1 = Error.
#
# ACCESSED GLOBAL VARIABLES:
#    None.
# =============================================================================
sub ProcessTurnoutFile {

   my($FileName, $Function, $TurnoutData) = @_;
   my($turnout, $rec);
   my(@fileData) = ();

   my(@keyList) = ('Pid','Addr','Port','Pos','Rate','Open','Middle','Close',
                   'MinPos','MaxPos','Id');

   &DisplayDebug(2, "ProcessTurnoutFile, Function: $Function   " .
                    "keyList: '@keyList'");

   if ($Function =~ m/^Read$/i) {
      if (-e $FileName) {
         if (&ReadFile($FileName, \@fileData)) {
            &DisplayWarning("ProcessTurnoutData, Using default " .
                            "turnout data.");
         }
         else {
            %$TurnoutData = ();
            foreach my $rec (@fileData) {
               next if ($rec =~ m/^\s*$/ or $rec =~ m/^#/);
               if ($rec =~ m/Turnout:\s*(\d+)/i) {
                  $turnout = sprintf("%2s",$1);
                  $$TurnoutData{$turnout}{'Pid'} = 0;
                  foreach my $key (@keyList) {
                     if ($key eq 'Id') {
                        if ($rec =~ m/$key:(.+)/) {
                           $$TurnoutData{$turnout}{$key} = &Trim($1);
                        }
                        else {
                           &DisplayWarning("ProcessTurnoutData, " .
                                           "'$key' not found: '$rec'");
                           next;
                        }
                     }
                     else {
                        if ($rec =~ m/$key:\s*(\d+)/) {
                           $$TurnoutData{$turnout}{$key} = $1;
                        }
                        else {
                           &DisplayWarning("ProcessTurnoutData, " .
                                           "'$key' not found: '$rec'");
                           next
                        }
                     }
                     &DisplayDebug(2, "ProcessTurnoutFile, " .
                                   "Turnout: $turnout   key: $key   value: " .
                                   "$$TurnoutData{$turnout}{$key}");
                  }
               }
               else {
                  &DisplayWarning("ProcessTurnoutData, 'Turnout' key " .
                                  "not found: '$rec'");
               }
            }
         }
         $rec = scalar keys %$TurnoutData;
         &DisplayDebug(1, "ProcessTurnoutFile, Function: $Function " .
                          "$rec turnout records.");
      }
      else {
         &DisplayWarning("ProcessTurnoutData: File not found: $FileName.");
         &DisplayWarning("ProcessTurnoutData: Using default turnout data.");
      }
   }
   elsif ($Function =~ m/^Write$/i) {
      push (@fileData, "# ===============================================");
      push (@fileData, "# Turnout data file. Loaded during program start.");
      push (@fileData, "# Edited values will be used upon next start. See");
      push (@fileData, "# DnB.pl 'Turnout Related Data' section for more ");
      push (@fileData, "# information.");
      push (@fileData, "# ===============================================");

      $rec = scalar keys %$TurnoutData;
      &DisplayDebug(1, "ProcessTurnoutFile, Function: $Function $rec " .
                       "turnout records.");

      foreach my $turnout (sort keys %$TurnoutData) {
         next if ($turnout =~ m/^\s*$/ or $turnout eq '00');
         $rec = join(":", "Turnout", $turnout);
         $$TurnoutData{$turnout}{'Pid'} = 0;
         foreach my $key (@keyList) {
            $rec = join(" ", $rec, join(":", $key,
                        $$TurnoutData{$turnout}{$key}));
         }
         push (@fileData, $rec);
         &DisplayDebug(2, "ProcessTurnoutFile, $Function: $rec");
      }
      &WriteFile($FileName, \@fileData);
   }
   else {
      &DisplayWarning("ProcessTurnoutData, Unsupported function: $Function");
   }
   return 0;
}

# ==============================================================================
# FUNCTION:  InitTurnouts
#
# DESCRIPTION:
#    Called once during DnB startup, this routine initializes all turnouts to
#    the PWM position specified in %TurnoutData. This ensures that all servo
#    driver board channels are synchronized to the %TurnoutData specified PWM
#    position.
#
#    A check of the %TurnoutData PWM values is performed since these values are
#    normally loaded from the user editable TurnoutDataFile. If an out-of-range
#    value is detected, initialization is aborted and an error is returned.
#
```

```
#    If optional data is specified, the servo is set to the specified PWM
#    position. This position is used for physical turnout point adjustment.
#
# CALLING SYNTAX:
#    $result = &InitTurnouts(\%ServoBoardAddress, \%TurnoutData, $Turnout,
#                            $Position);
#
# ARGUMENTS:
#    $ServoBoardAddress      Pointer to %ServoBoardAddress hash.
#    $TurnoutData            Pointer to %TurnoutData hash.
#    $Turnout                Optional; turnout to position.
#    $Position               Optional; position to set.
#
# RETURNED VALUES:
#    0 = Success,  1 = Error.
#
# ACCESSED GLOBAL VARIABLES:
#    None.
# =============================================================================
sub InitTurnouts {
   my($ServoBoardAddress, $TurnoutData, $Turnout, $Position) = @_;
   my($board, $pwm);
   my($min,$max) = (300,900);                 # Absolute PWM values.
   my($rmin,$rmax) = (1,850);                 # Absolute Rate values.
   my($fail) = 0;

   # Processing for -o, -m, and -c CLI options.
   if ($Turnout ne '') {
      $Turnout = "0${Turnout}" if (length($Turnout) == 1);
      if ($Position ne 'Open' and $Position ne 'Close') {
         $Position = 'Middle';
      }
   }

   # Validate the %TurnoutData PWM values.
   &DisplayMessage("Validate turnout PWM working values ...");
   foreach my $tNmbr (sort keys %$TurnoutData) {
      next if ($tNmbr eq '00');   # Skip temperature adjustment data.
      foreach my $pos ('MinPos','MaxPos','Open','Middle','Close','Pos') {
         $pwm = $$TurnoutData{$tNmbr}{$pos};
         if ($pwm < $min or $pwm > $max) {
            &DisplayError("InitTurnouts, turnout $tNmbr $pos " .
                          "value out of range: $pwm");
            $fail = 1;
         }
         elsif ($pwm < $$TurnoutData{$tNmbr}{'MinPos'} or
            $pwm > $$TurnoutData{$tNmbr}{'MaxPos'}) {
            &DisplayError("InitTurnouts, turnout $tNmbr $pos " .
                          "value outside of min/max limit: $pwm");
            $fail = 1;
         }
      }
      $pwm = $$TurnoutData{$tNmbr}{'Rate'};
      if ($pwm < $rmin or $pwm > $rmax) {
         &DisplayError("InitTurnouts, turnout $tNmbr Rate " .
                       "value out of range: $pwm");
         $fail = 1;
      }
   }
   return 1 if ($fail == 1);    # Error return if failure.

   # Initialize servo channel on the driver boards.
   for ($board = 1; $board <= scalar keys(%$ServoBoardAddress); $board++) {
      if ($$ServoBoardAddress{$board} == 0) {
         &DisplayDebug(1, "InitTurnouts, Skip board $board " .
                          "I2C_Address 0, code debug.");
         next;
      }
      &DisplayMessage("Initializing turnout I2C board $board ...");
      return 1 if (&I2C_InitServoDriver($board, $$ServoBoardAddress{$board}));

      &DisplayMessage("Initializing turnout positions on board $board ...");

      foreach my $tNmbr (sort keys %$TurnoutData) {
         next if ($tNmbr eq '00');   # Skip temperature adjustment data.
         if ($$TurnoutData{$tNmbr}{'Addr'} == $$ServoBoardAddress{$board}) {
            if ($Turnout eq '00' or $Turnout eq $tNmbr) {
               $$TurnoutData{$tNmbr}{'Pos'} = $$TurnoutData{$tNmbr}{$Position};
            }

            if (&SetTurnoutPosition($$TurnoutData{$tNmbr}{'Pos'}, $tNmbr,
                                    $TurnoutData)) {
               &DisplayWarning("InitTurnouts, Failed to set " .
                               "turnout.   board $board   Turnout: $tNmbr" .
                               "Position: $$TurnoutData{$tNmbr}{'Pos'}");
               $fail = 1;
            }

            $$TurnoutData{$tNmbr}{'Pid'} = 0;  # Ensure the Pid value is 0.
            sleep 0.1;                         # Delay so we don't overtax
                                               # the servo power supply.
         }
      }
      &DisplayMessage("All board $board turnouts initialized.");
   }
   if ($Turnout ne '') {
      if ($Turnout eq '00') {
         &DisplayMessage("All turnouts set to $Position position.");
      }
      else {
         &DisplayMessage("Turnout $Turnout set to $Position position.");
      }
   }
   return 1 if ($fail == 1);    # Error return if failure.
   return 0;
}

# =============================================================================
# FUNCTION:  MoveTurnout
#
# DESCRIPTION:
#    This routine moves the turnout servo using the specified data. It is used
#    to perform a slow motion position change. This is done by forking to a
#    child process and calling SetTurnoutPosition 50 times a second until the
#    move is complete. Each call positions the turnout servo toward the final
#    position by a move step amount ('Rate'/50). Once the move is completed,
#    the turnout position is updated in the TurnoutData hash and the child
#    exits. A 'Rate' value of 450 positions the turnout from Open (350) to
#    Close (850) in about 1.1 seconds.
#
```

```
# CALLING SYNTAX:
#    $result = &MoveTurnout($Function, $TurnoutNmbr, \%TurnoutData);
#
# ARGUMENTS:
#    $Function       'Open', 'Middle', or 'Close'.
#    $TurnoutNmbr    Turnout number; two digit hash index.
#    $TurnoutData    Pointer to TurnoutData hash.
#
# RETURNED VALUES:
#    0 = Success,  1 = Error, 2 = Already in position.
#
# ACCESSED GLOBAL VARIABLES:
#    None.
# =============================================================================
sub MoveTurnout {
   my($Function, $TurnoutNmbr, $TurnoutData) = @_;
   my($result, $pwmCurrent, $pwmFinal, $moveRate, $moveStep, $pid, $adjust);
   my($noAdj);
   my($timeout) = 40;    # Wait 10 seconds (40/.25) for move to complete.

   &DisplayDebug(2, "MoveTurnout, Entry ...   $Function $TurnoutNmbr");

   if ($TurnoutNmbr ne "") {
      if ($Function =~ m/Open/i) {
         $pwmFinal = $$TurnoutData{$TurnoutNmbr}{'Open'};
      }
      elsif ($Function =~ m/Middle/i) {
         $pwmFinal = $$TurnoutData{$TurnoutNmbr}{'Middle'};
      }
      elsif ($Function =~ m/Close/i) {
         $pwmFinal = $$TurnoutData{$TurnoutNmbr}{'Close'};
      }
      else {
         &DisplayError("MoveTurnout, invalid function: '$Function'");
         return 1;
      }

      # If gate or semaphore servo, adjust $pwmFinal for temperature.
      if ($$TurnoutData{$TurnoutNmbr}{'Id'} =~ m/semaphore/i or
          $$TurnoutData{$TurnoutNmbr}{'Id'} =~ m/gate/i) {
         if ($$TurnoutData{'00'}{'Temperature'} > 0 and
             $$TurnoutData{'00'}{'Temperature'} < 38) {
            $noAdj = $pwmFinal;     # Used only for debug message.
            #  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33 35 37  degree C
            # -8 -7 -6 -5 -4 -3 -2 -1  0 +1 +2 +3 +4 +5 +6 +7 +8  step adjust
            # Change divisor (-2) to increase/decrease overall step count.
            # Change constant (21) to shift center point temperature.
            # Note: TurnoutData MinPos and MaxPos will limit adjustment if
            #       set too close to Open/Close value.
            $adjust = int((21 - $$TurnoutData{'00'}{'Temperature'}) / -2);
            &DisplayDebug(1, "MoveTurnout, servo: $TurnoutNmbr   " .
                             "adjust: $adjust");

            # Application of adjustment is dependent on close direction.
            if ($$TurnoutData{$TurnoutNmbr}{'Open'} >
                $$TurnoutData{$TurnoutNmbr}{'Close'}) {
               $pwmFinal += $adjust;
            }
            else {
               $pwmFinal -= $adjust;
            }
            &DisplayDebug(1, "MoveTurnout, noAdj: $noAdj   adjusted: $pwmFinal");
         }
      }

      # Make sure the requested move will not exceed a min/max limit.
      $pwmFinal = $$TurnoutData{$TurnoutNmbr}{'MinPos'}
                  if ($pwmFinal < $$TurnoutData{$TurnoutNmbr}{'MinPos'});
      $pwmFinal = $$TurnoutData{$TurnoutNmbr}{'MaxPos'}
                  if ($pwmFinal > $$TurnoutData{$TurnoutNmbr}{'MaxPos'});

      # Check and wait for turnout to be idle.
      while ($$TurnoutData{$TurnoutNmbr}{'Pid'} > 0 and $timeout > 0) {
         if (($timeout % 4) == 0) {
            &DisplayDebug(2, "MoveTurnout, waiting for previous move " .
                             "to complete. timeout: $timeout   Pid: " .
                             "$$TurnoutData{$TurnoutNmbr}{'Pid'}   Pos: " .
                             "$$TurnoutData{$TurnoutNmbr}{'Pos'}");
         }
         $timeout--;
         sleep 0.25;                # Wait quarter sec.
      }

      # Abort turnout move if still active.
      if ($$TurnoutData{$TurnoutNmbr}{Pid} > 0) {
         &DisplayError("MoveTurnout, Turnout $TurnoutNmbr, Previous " .
                       "move still in progress, pid: " .
                       "$$TurnoutData{$TurnoutNmbr}{'Pid'}.");

         # Check if the process is running, $result == 0. If so, kill it.
         # Cleanup state data and continue new turnout move.
         $result = waitpid($$TurnoutData{$TurnoutNmbr}{'Pid'}, WNOHANG);
         system("kill -9 $$TurnoutData{$TurnoutNmbr}{'Pid'}") if ($result == 0);
         $$TurnoutData{$TurnoutNmbr}{'Pid'} = 0;
      }

      $pwmCurrent = $$TurnoutData{$TurnoutNmbr}{'Pos'};
      if ($pwmCurrent == $pwmFinal) {           # Done if already in position.
         &DisplayDebug(2, "MoveTurnout, $TurnoutNmbr already in " .
                          "requested position: $pwmFinal");
         return 2;
      }

      $moveRate = $$TurnoutData{$TurnoutNmbr}{'Rate'};

      if ($moveRate > 0) {
         # Fork program to complete the move. Use Forks::Super which is a go
         # between the parent and child. It has a function for writing child
         # data back to the main program using child STDOUT and STDERR. It is
         # not necessary to 'reap' the child when using Forks::Super. Also,
         # SIG{CHILD} should not be set by this program. It is set/used by
         # Forks::Super. Do no other printing, including debug output.
         #
         # STDERR: move complete. $TurnoutData{<tNmbr>}{'Pid'} set to 0.
         # STDOUT: new turnout position. $TurnoutData{<tNmbr>}{'Pos'}.

         &DisplayDebug(2, "MoveTurnout, pre-fork: $Function " .
                          "$TurnoutNmbr   pwmCurrent: $pwmCurrent" .
                          "   pwmFinal: $pwmFinal   moveRate: $moveRate");

         $pid = fork { os_priority => 1,
                       stdout => \$$TurnoutData{$TurnoutNmbr}{'Pos'},
                       stderr => \$$TurnoutData{$TurnoutNmbr}{'Pid'} };
         if (!defined($pid)) {
            &DisplayError("TurnoutChildProcess, Failed to create " .
                          "child process. $!");
            return 1;
         }
#----------
         elsif ($pid == 0) {          # fork returned 0, so this is the child
            $moveStep = $moveRate/50;              # Step increment
            while ($pwmCurrent != $pwmFinal) {
               if ($pwmCurrent < $pwmFinal) {      # Determine move direction
                  $pwmCurrent += $moveStep;
                  $pwmCurrent = $pwmFinal if ($pwmCurrent > $pwmFinal);
               }
               else {
                  $pwmCurrent -= $moveStep;
                  $pwmCurrent = $pwmFinal if ($pwmCurrent < $pwmFinal);
               }

               if (&SetTurnoutPosition($pwmCurrent, $TurnoutNmbr, $TurnoutData)) {
                  # Retain previous pwmCurrent in Pos if error is returned.
                  print STDERR 0;            # Clear Pid, move has completed.
                  exit(1);                   # Starting position is retained.
               }
               sleep 0.02;
            }
            print STDOUT $pwmCurrent;        # Store position of turnout
            print STDERR 0;                  # Clear Pid, move has completed.
            exit(0);
         }
#----------
         $$TurnoutData{$TurnoutNmbr}{'Pid'} = $pid;  # Parent: Move in-progress.
         &DisplayDebug(1, "MoveTurnout, $Function $TurnoutNmbr " .
                          "forked pid: $$TurnoutData{$TurnoutNmbr}{'Pid'}");
      }
      else {
         &DisplayWarning("MoveTurnout, Rate value must be greater than 0.");
         return 1;
      }
   }
   else {
      &DisplayError("MoveTurnout, invalid turnout number: $TurnoutNmbr");
      return 1;
   }
   return 0;
}

# =============================================================================
# FUNCTION:  SetTurnoutPosition
#
# DESCRIPTION:
#    This routine sets the turnout servo using the specified data. This
#    routune writes the I2C interface with the needed command bytes.
#
#    This routine checks the Position value to provide some servo protection
#    due to a possible program runtime error.
#
# CALLING SYNTAX:
```

```
#    $result = &SetTurnoutPosition($Position, $TurnoutNmbr, \%TurnoutData);
#
# ARGUMENTS:
#    $Position       PWM position to set.
#    $TurnoutNmbr    Turnout number.
#    $TurnoutData    Pointer to TurnoutData hash.
#
# RETURNED VALUES:
#    0 = Success,  1 = Error.
#
# ACCESSED GLOBAL VARIABLES:
#    None.
# ==============================================================================
sub SetTurnoutPosition {
   my($Position, $TurnoutNmbr, $TurnoutData) = @_;
   my($driver, $reg_start, $reg_data_on, $reg_data_off);
   my(@data) = ();

   # The MoveTurnout subroutine uses STDOUT and STDERR to report final turnout
   # position to the parent process. Debug messaging must be commented out if
   # not doing code debug. Otherwise, TurnoutDataFile.txt will be corrupted
   # when Ctrl+C is used.

   # &DisplayDebug(2, "SetTurnoutPosition, $TurnoutNmbr - $Position");

   if (exists($$TurnoutData{$TurnoutNmbr})) {
      $Position = int($Position);
      if ($Position < $$TurnoutData{$TurnoutNmbr}{'MinPos'}) {
         $Position = $$TurnoutData{$TurnoutNmbr}{'MinPos'};
         # &DisplayWarning("SetTurnoutPosition, Turnout $TurnoutNmbr " .
         #                 "PWM value beyond MinPos limit. Set to " .
         #                 "MinPos $Position");
      }
      if ($Position > $$TurnoutData{$TurnoutNmbr}{'MaxPos'}) {
         $Position = $$TurnoutData{$TurnoutNmbr}{'MaxPos'};
         # &DisplayWarning("SetTurnoutPosition, Turnout $TurnoutNmbr " .
         #                 "PWM value beyond MaxPos limit. Set to ".
         #                 "MaxPos $Position");
      }

      $reg_start = (($$TurnoutData{$TurnoutNmbr}{'Port'} % 16) * 4) + 6;

      # Stagger pulse start (* 10) to minimuze power drops.
      $reg_data_on = $$TurnoutData{$TurnoutNmbr}{'Port'} * 10;
      push (@data, ($reg_data_on & 0xFF));            # on_L
      push (@data, (($reg_data_off >> 8) & 0x0F));    # on_H
      $reg_data_off = $reg_data_on + $Position;
      push (@data, ($reg_data_off & 0xFF));           # off_L
      push (@data, (($reg_data_off >> 8) & 0x0F));    # off_H

      $driver = RPi::I2C->new($$TurnoutData{$TurnoutNmbr}{'Addr'});
      unless ($driver->check_device($$TurnoutData{$TurnoutNmbr}{'Addr'})) {
         &DisplayError("SetTurnoutPosition, Failed to initialize " .
                       "I2C address: " .
                       sprintf("%.2x",$$TurnoutData{$TurnoutNmbr}{'Addr'}));
         return 1;
      }
      $driver->write_block(\@data, $reg_start);
      undef($driver);
   }
   else {
      &DisplayError("SetTurnoutPosition, invalid turnout number: $TurnoutNmbr");
      return 1;
   }
   return 0;
}

# =============================================================================
# FUNCTION:  GetTemperature
#
# DESCRIPTION:
#    This routine gets the current temperature value in degrees Celsius from
#    the DS18B20 sensor attached to GPIO4. A timeout variable is also set to
#    facilitate future calls to this code.
#
#    The DS18B20 sensor is a 1-wire protocol device that is interfaced using
#    raspbian modprobe. The device must be configured external to this program.
#    Add the following.
#
#    sudo nano /boot/config.txt
#       dtoverlay=w1-gpio
#
#    sudo nano /etc/modules
#       w1-gpio
#       w1-therm
#
#    Reboot RPi.
#
#    Then use 'ls /sys/bus/w1/devices' to list the unique device ID and replace
#    <sensorId> in the $sensor variable below.
#
#    If a DS18B20 sensor is not present or misconfigured, safe values are set
#    in the TurnoutData hash.
#
# Amnient temperature accuracy is affected by the sensor's proximity to the
# warm circuit board electronics. The $calibration variable adjusts the
# returned temperature value based on comparison with thermometer measurement.
#
# Use a digital thermometer to measure the layout benchwork temperature and
# compare it to the temperature value displayed on the console during DnB.pl
# startup. Enter an appropriate adjustment value into $calibration.
#
# CALLING SYNTAX:
#    $result = &GetTemperature(\%TurnoutData);
#
# ARGUMENTS:
#    $TurnoutData    Pointer to TurnoutData hash.
#
# RETURNED VALUES:
#    0 = Error, non-zero = temperature.
#
# ACCESSED GLOBAL VARIABLES:
#    None.
# =============================================================================
sub GetTemperature {
   my($TurnoutData) = @_;
   my($temp);
   #              /sys/bus/w1/devices/<sensorId>/w1_slave
   my($sensor) = '/sys/bus/w1/devices/28-030197944687/w1_slave';
   my($calibration) = 1.837;    # Centigrade value!
   my($temperature) = 0;

   if (-e $sensor) {
      my $result = `cat $sensor`;
      if ($result =~ m/t=(\d+)/) {
         $temp = $1 / 1000;
         if ($temp > 0 and $temp < 38) {
            $temperature = $temp - $calibration;
         }
         else {
            &DisplayError(1, "GetTemperature, Invalid temperature: $temperature");
         }
      }
      else {
         &DisplayDebug(1, "GetTemperature, Temperature value not parsed.");
      }
   }
   else {
      &DisplayDebug(1, "GetTemperature, DS18B20 sensor is not configured.");
   }
   $$TurnoutData{'00'}{'Temperature'} = $temperature;
   $$TurnoutData{'00'}{'Timeout'} = time + 300;
   return $temperature;
}

# =============================================================================
# FUNCTION:  TestServoAdjust
#
# DESCRIPTION:
#    This routine cycles the specified turnout range between the open and
#    closed positions.
#
# CALLING SYNTAX:
#    $result = &TestServoAdjust($Param, \%TurnoutData);
#
# ARGUMENTS:
#    $Param           Servo number and temperatures. -w Tx[p]:t1,t2,...
#    $TurnoutData     Pointer to TurnoutData hash.
#
# RETURNED VALUES:
#    0 = Success,  1 = Error.
#
# ACCESSED GLOBAL VARIABLES:
#    $main::MainRun
# =============================================================================
sub TestServoAdjust {

   my($Param, $TurnoutData) = @_;
   my($servo, $position, $temp, $pos, $origPos, $sndFlag, $result);
   my(@positions, @temperatures);

   &DisplayDebug(1, "TestServoAdjust, Entry ... Param: '$Param'");
   if ($Param =~ m/^(\d+)(\D*):(.+)/) {
      $servo = $1;
      $position = lc($2);
      @temperatures = split(',', $3);

      # Validate input parameters.
      $servo = "0${servo}" if (length($servo) == 1);
      unless (exists($$TurnoutData{$servo})) {
         &DisplayError("TestServoAdjust, invalid servo number: $servo");
         return 1;
      }
      if ($position eq '') {
         @positions = ('Open','Middle','Close');
      }
      elsif ($position =~ m/o/) {
         @positions = ('Open');
      }
      elsif ($position =~ m/m/) {
         @positions = ('Middle');
      }
      elsif ($position =~ m/c/) {
         @positions = ('Close');
      }
      else {
         &DisplayError("TestServoAdjust, invalid position: $position");
         return 1;
      }
      foreach my $temp (@temperatures) {
         $temp = &Trim($temp);
         unless ($temp > 0 and $temp < 38) {
            &DisplayError("TestServoAdjust, invalid temperature: $temp");
            return 1;
         }
      }

      # Save current servo position for later restoration.
      foreach my $pos ('Open','Middle','Close') {
         if ($$TurnoutData{$servo}{$pos} eq $$TurnoutData{$servo}{'Pos'}) {
            $origPos = $pos;
            last;
         }
      }

      # Start testing.
      while ($main::MainRun) {
         foreach my $pos (@positions) {
            $sndFlag = 1;
            foreach my $temp (@temperatures) {
               $$TurnoutData{'00'}{'Temperature'} = $temp;
               $result = &MoveTurnout($pos, $servo, $TurnoutData);
               &DisplayDebug(1, "TestServoAdjust, pos: $pos   servo: '$servo' (" .
                                $$TurnoutData{$servo}{'Id'} . ")   " .
                                "temp: $temp   result: $result");
               # Sound tone.
               if ($sndFlag eq 1) {
                  &PlaySound("C.wav");
                  $sndFlag = 0;
               }
               else {
                  &PlaySound("E.wav");
               }
               # Wait for move to complete.
               while ($$TurnoutData{$servo}{'Pid'}) {
                  sleep 0.25;
               }
               last if ($main::MainRun == 0);
               sleep 2;  # Intra-temperature delay
            }
            last if ($main::MainRun == 0);
         }
      }

      # Restore original servo position.
      $$TurnoutData{'00'}{'Temperature'} = 0;
      $result = &MoveTurnout($origPos, $servo, $TurnoutData);
      while ($$TurnoutData{$servo}{'Pid'}) {
         sleep 0.25;
      }
   }
   else {
      &DisplayError("TestServoAdjust, invalid parameters: '$Param'");
      return 1;
   }
   return 0;
}

# =============================================================================
# FUNCTION:  TestTurnouts
#
# DESCRIPTION:
#    This routine cycles the specified turnout range between the open and
#    closed positions.
#
# CALLING SYNTAX:
#    $result = &TestTurnouts($Range, \%TurnoutData);
#
# ARGUMENTS:
#    $Range           Turnout number or range to use.
#    $TurnoutData     Pointer to TurnoutData hash.
#
# RETURNED VALUES:
#    0 = Success,  1 = Error.
#
# ACCESSED GLOBAL VARIABLES:
#    $main::MainRun
# =============================================================================
sub TestTurnouts {

   my($Range, $TurnoutData) = @_;
   my($moveResult, $turnout, $start, $end, $nmbr, $oper, $pid, $cnt,
      @turnoutNumbers, @inProgress, $position);
   my($cntTurnout) = scalar keys %$TurnoutData;
   my(%operation) = (1 => 'Open ', 2 => 'Close');
   my(@turnoutList) = ();
   my($random, $wait) = (0, 0);

   &DisplayDebug(1, "TestTurnouts, Entry ... Range: '$Range'   " .
                    "cntTurnout: $cntTurnout");

   # ==============================
   # Set specified position and exit.

   if ($Range =~ m/^(Open):(\d+)/i or $Range =~ m/^(Close):(\d+)/i or
       $Range =~ m/^(Middle):(\d+)/i) {
      $position = ucfirst(lc $1);
      $turnout = $2;
      $turnout = "0${turnout}" if (length($turnout) == 1);

      # The %TurnoutData Id string must contain the word turnout.
      if ($$TurnoutData{$turnout}{'Id'} =~ m/turnout/) {
         &MoveTurnout($position, $turnout, $TurnoutData);
         &DisplayMessage("Turnout $turnout set to '$position'.");
      }
      else {
         &DisplayError("TestTurnouts, invalid turnout number: $turnout");
      }
      exit(0);
   }
   elsif ($Range =~ m/^(Open)$/i or $Range =~ m/^(Close)$/i or
          $Range =~ m/^(Middle)$/i) {
      $position = ucfirst(lc $1);

      # The %TurnoutData Id string must contain the word turnout.
      foreach my $turnout (sort keys %$TurnoutData) {
         if ($$TurnoutData{$turnout}{'Id'} =~ m/turnout/) {
            &MoveTurnout($position, $turnout, $TurnoutData);
            &DisplayDebug(1, "TestTurnouts, turnout: $turnout set " .
                             "to $position");
         }
      }
      &DisplayMessage("All turnouts set to '$position'.");
      exit(0);
   }

   # ==============================
   # Process special modifiers and then setup for looped testing.

   if ($Range =~ m/r/i) {
      $random = 1;
      $Range =~ s/r//i;
   }
   if ($Range =~ m/w/i) {
      $wait = 1;
      $Range =~ s/w//i;
   }

   if ($Range =~ m/(\d+):(\d+)/) {   # Range specified.
      $start = $1;
      $end = $2;
      if ($start > $end or $start <= 0 or $start > $cntTurnout or $end <= 0 or
          $end > $cntTurnout) {
         &DisplayError("TestTurnouts, invalid turnout range: '$Range'" .
                       "   cntTurnout: $cntTurnout");
         return 1;
      }
      for ($turnout = $start; $turnout <= $end; $turnout++) {
         push (@turnoutList, $turnout);
      }
   }
   else {
      @turnoutList = split(",", $Range);
   }
   &DisplayDebug(1, "TestTurnouts, random: $random   wait: $wait   " .
                    "turnoutList: '@turnoutList'");

   # Identify the servos being used for turnouts. The %TurnoutData Id string
   # must contain the word turnout.
   foreach my $key (sort keys %$TurnoutData) {
      if ($$TurnoutData{$key}{'Id'} =~ m/turnout/) {
         push (@turnoutNumbers, $key);
      }
   }

   $oper = 'Open  ';
   while ($main::MainRun) {
      # For random testing, we randomize the turnoutNumbers list and also the
      # Open/Close operation. For non-random, Open and then Close the turnouts
      # in the specified order.
      &ShuffleArray(\@turnoutNumbers) if ($random == 1);

      foreach my $turnout (@turnoutNumbers) {
         return 0 unless ($main::MainRun);
         $nmbr = $turnout;
         $nmbr =~ s/^0//;
         if (grep /^$nmbr$/, @turnoutList) {  # Move turnout if on the list.
            $oper = $operation{(int(rand(2))+1)} if ($random == 1);
            if ($#inProgress < 0) {
               &DisplayMessage("TestTurnouts, $oper $turnout   Concurrent " .
                               "moves: none");
            }
            else {
               &DisplayMessage("TestTurnouts, $oper $turnout   Concurrent " .
                               "moves: @inProgress");
            }
            $moveResult = &MoveTurnout($oper, $turnout, $TurnoutData);
            return 1 if ($moveResult == 1);
            if ($moveResult == 2) {
               &DisplayDebug(2, "TestTurnouts, MoveTurnout $turnout returned " .
                                "already in position.");
            }
            elsif ($moveResult == 0) {
               if ($wait == 1) {
                  $cnt = 20;
                  while ($$TurnoutData{$turnout}{'Pid'}) {
                     if ($cnt == 0) {
                        &DisplayError("TestTurnouts, timeout waiting for " .
                                      "turnout $turnout to complete positioning.");
                        return 1;
                     }
                     &DisplayDebug(2, "TestTurnouts, waiting for " .
                                      "pid: $$TurnoutData{$turnout}{'Pid'}");
                     sleep 0.5;
                     $cnt--;
                  }
                  &DisplayDebug(2, "TestTurnouts, Turnout $turnout new position: " .
                                   "$$TurnoutData{$turnout}{'Pos'}");
               }
            }
            @inProgress = ();
            foreach my $key (sort keys(%$TurnoutData)) {
               push (@inProgress, $key) if ($$TurnoutData{$key}{'Pid'} != 0);
            }
            sleep 0.05 unless ($moveResult == 2);
         }
      }

      if ($random == 0) {   # Change if doing sequential testing.
         if ($oper =~ m/Open/) {
            $oper = 'Close ';
         }
         else {
            $oper = 'Open  ';
         }
      }
      sleep 2;
   }
   return 0;
}

return 1;
```