```
# ============================================================================
# FILE: DnB_Sensor.pm                                              9/20/2020
#
# SERVICES:  DnB SENSOR FUNCTIONS
#
# DESCRIPTION:
#    This perl module provides sensor related functions used by the DnB model
#    railroad control program.
#
# PERL VERSION: 5.24.1
#
# ============================================================================
use strict;
# ----------------------------------------------------------------------------
# Package Declaration
# ----------------------------------------------------------------------------
package DnB_Sensor;
require Exporter;
our @ISA = qw(Exporter);

our @EXPORT = qw(
   I2C_InitSensorDriver
   KeypadChildProcess
   ButtonChildProcess
   PositionChildProcess
   GetSensorBit
   ReadKeypad
   GetButton
   TestSensorBits
   TestSensorTones
   TestKeypad
);

use DnB_Message;
use Time::HiRes qw(gettimeofday sleep);

# ============================================================================
# FUNCTION:  I2C_InitSensorDriver
#
# DESCRIPTION:
#    This routine initializes the sensor I2C driver board on the DnB model
#    railroad. It sets parameters that are common to all sensor ports. The
#    I/O PI Plus board utilizes two MCP23017 chips. Each chip has two 8 bit
#    ports of configurable GPIO pins. Each chip is configured based on the
#    values in the %SensorChip hash.
#
#    Chip 3 is initialized for a 'Storm K Range' 4x4 keypad. MCP23017 GPIO
#    pins are direct connected as follows. Row (letter) GPIOs are set as
#    input + pullup. Columns set as outputs.
#
#    The %SensorChip{chip}{'Obj'} hash key is written with the driver object
#    pointer for use in sensor data reading.
#
#       Keypad pin:      1 2 3 4 5 6 7 8
#       Keypad col/row:  A B 1 2 3 4 D C
#       GPIOA pin:       3 4 5 6 7 8 9 10
#       GPIOA bit:       0 1 2 3 4 5 6 7
#       GPIODIRA:        1 1 0 0 0 0 1 1     1 = Input, 0 = Output
#
# CALLING SYNTAX:
```

```
#    $result = &I2C_InitSensorDriver($ChipNmbr, \%MCP23017, \%SensorChip);
#
# ARGUMENTS:
#    $ChipNmbr             Chip number being initialized.
#    $MCP23017             Pointer to %MCP23017 internal register definitions
#    $SensorChip           Pointer to %SensorChip hash.
#
# RETURNED VALUES:
#    0 = Success,  1 = Error.
#
# ACCESSED GLOBAL VARIABLES:
#    None.
# ==========================================================================
sub I2C_InitSensorDriver {

   my($ChipNmbr, $MCP23017, $SensorChip) = @_;
   my($driver);
   my(@temp) = ();

   &DisplayDebug(2, "I2C_InitSensorDriver, ChipNmbr: $ChipNmbr   I2C_Address: " .
                    sprintf("0x%.2X",$$SensorChip{$ChipNmbr}{'Addr'}));

   $driver = RPi::I2C->new($$SensorChip{$ChipNmbr}{'Addr'});
   unless ($driver->check_device($$SensorChip{$ChipNmbr}{'Addr'})) {
      &DisplayError("I2C_InitSensorDriver, Failed to initialize I2C address: " .
                    sprintf("0x%.2X",$$SensorChip{$ChipNmbr}{'Addr'}) .
                    " - $!");
      return 1;
   }
   $$SensorChip{$ChipNmbr}{'Obj'} = $driver;

   # Set MCP23017 BANK (bit7) = 0 (sets MCP23017 register addresses)
   $driver->write_byte(0x00, $$MCP23017{'IOCON'});

   # Set port direction bits.
   $driver->write_byte($$SensorChip{$ChipNmbr}{'DirA'}, $$MCP23017{'IODIRA'});
   $driver->write_byte($$SensorChip{$ChipNmbr}{'DirB'}, $$MCP23017{'IODIRB'});

   # Set port polarity bits.
   $driver->write_byte($$SensorChip{$ChipNmbr}{'PolA'}, $$MCP23017{'IOPOLA'});
   $driver->write_byte($$SensorChip{$ChipNmbr}{'PolB'}, $$MCP23017{'IOPOLB'});

   # Set port pullup enable bits.
   $driver->write_byte($$SensorChip{$ChipNmbr}{'PupA'}, $$MCP23017{'GPPUA'});
   $driver->write_byte($$SensorChip{$ChipNmbr}{'PupB'}, $$MCP23017{'GPPUB'});

   # Build temporary array for debug output.
   push(@temp, $driver->read_byte($$MCP23017{'IOCON'}));   # Get current IOCON.
   push(@temp, $driver->read_byte($$MCP23017{'IODIRA'}));  # Get current IODIRA.
   push(@temp, $driver->read_byte($$MCP23017{'IODIRB'}));  # Get current IODIRB.
   push(@temp, $driver->read_byte($$MCP23017{'IOPOLA'}));  # Get current IOPOLA.
   push(@temp, $driver->read_byte($$MCP23017{'IOPOLB'}));  # Get current IOPOLB.

   &DisplayDebug(2, "I2C_InitSensorDriver - Initialized IODIRA " .
                    "IODIRB IOPOLA IOPOLB IOCON: " .
                    sprintf("0x%0.2X 0x%0.2X 0x%0.2X 0x%0.2X 0x%0.2X", @temp));

# Debug code.
#   while (1) {
#      $message = "I2C Address: " . sprintf("0x%.2X",$$SensorChip{$ChipNmbr}
```

```
121 #                                          {'Addr'});
122 #      $message = $message . "   GPIOB: " . sprintf("%0.8b",
123 #                                       $driver->read_byte($$MCP23017{'GPIOB'}));
124 #      $message = $message . "   GPIOA: " .
125 #                      sprintf("%0.8b", $driver->read_byte($$MCP23017{'GPIOA'}));
126 #      &DisplayMessage("I2C_InitSensorDriver - $message");
127 #      sleep 1;
128 #   }
129 #   exit(0);
130
131     return 0;
132 }
133
134 # ============================================================================
135 # FUNCTION:  KeypadChildProcess
136 #
137 # DESCRIPTION:
138 #    This routine is launched as a child process during main program startup
139 #    and is used to return user input from the 'Storm K Range' 4x4 button
140 #    keypad. This keypad is connected to a MCP23017 port as follows.
141 #
142 #       row/col   1   2   3   4
143 #                 |   |   |   |
144 #         A ------0---1---2---3--
145 #                 |   |   |   |
146 #         B ------4---5---6---7--
147 #                 |   |   |   |
148 #         C ------8---9---A---B--
149 #                 |   |   |   |
150 #         D ------C---D---E---F--
151 #                 |   |   |   |
152 #
153 #       Keypad pin:      1 2 3 4 5 6 7 8
154 #       Keypad col/row:  A B 1 2 3 4 D C
155 #       GPIOA pin:       3 4 5 6 7 8 9 10
156 #       GPIOA bit:       0 1 2 3 4 5 6 7
157 #       GPIODIRA:        1 1 0 0 0 0 1 1     1 = Input, 0 = Output
158 #
159 #    A dedicated child process is used to improve the reliability of keypad
160 #    entries. Forks::Super is used between the parent and child to read data
161 #    from the child's STDERR filehandle. Do no other output to STDERR within
162 #    this routine. DisplayMessage and DisplayDebug are permitted since they
163 #    use STDOUT for messaging.
164 #
165 #    The %KeypadData hash provides keypad specific data and state information.
166 #    Data is accessed using a hash index specified in the $Keypad variable.
167 #
168 #    The %cols hash holds 4 values, each has one of the keypad column driver
169 #    bits low. These bits are configured as outputs by I2C_InitSensorDriver.
170 #    The %col hash keys map to the %matrix hash primary keys.
171 #
172 #    The %matrix hash contains the resulting button value for each of the 16
173 #    combinations of col/row. I2C_InitSensorDriver configures the input pins
174 #    with pullup enabled which results in a value of 0xC3 when no button is
175 #    pressed. The hash secondary key corresponds to 0xC3 with one of the input
176 #    bits low. Input is ignored if multiple buttons are pressed.
177 #
178 #    Note that the physical rotational orientation of the keypad, that is the
179 #    keys that are the top row, will necessitate changes to the %matrix hash
180 #    values. Current matrix values are for the orientation with the keypad
```

```
#    connector at the 6 o'clock position.
#
# CALLING SYNTAX:
#    $KeypadChildPid = fork {os_priority => 4, sub => \&KeypadChildProcess,
#                            child_fh => "err socket",
#                            args => [ $Keypad, \%KeypadData, \%MCP23017,
#                                      \%SensorChip ] };
#
#    $read_key = Forks::Super::read_stderr($KeypadChildPid);
#
# ARGUMENTS:
#    $Keypad               KeypadData entry to use.
#    $KeypadData           Pointer to %KeypadData hash.
#    $MCP23017             Pointer to MCP23017 internal register definitions
#    $SensorChip           Pointer to %SensorChip hash.
#
# RETURNED VALUES:
#    0-F = Pressed button via read_stderr.
#
# ACCESSED GLOBAL VARIABLES:
#    $main::ChildName
# ============================================================================
sub KeypadChildProcess {
   my($Keypad, $KeypadData, $MCP23017, $SensorChip) = @_;
   my($row, $button);
   my($chip) = $$KeypadData{$Keypad}{'Chip'};
   my(%cols) = (1 => 0xFB, 2 => 0xF7, 3 => 0xEF, 4 => 0xDF);
   my(%matrix) = (1 => { 0xC2 => '0', 0xC1 => '4', 0x43 => '8', 0x83 => 'C'},
                  2 => { 0xC2 => '1', 0xC1 => '5', 0x43 => '9', 0x83 => 'D'},
                  3 => { 0xC2 => '2', 0xC1 => '6', 0x43 => 'A', 0x83 => 'E'},
                  4 => { 0xC2 => '3', 0xC1 => '7', 0x43 => 'B', 0x83 => 'F'});

   $main::ChildName = 'KeypadChildProcess';
   &DisplayMessage("KeypadChildProcess started.");
   &DisplayDebug(2, "Keypad: $Keypad   chip: $chip");

   if ($$SensorChip{$chip}{'Obj'} == 0) {
      &DisplayMessage("*** error: KeypadChildProcess, No SensorChip object " .
                      "for chip $chip. Call I2C_InitSensorDriver routine first.");
      &DisplayMessage("KeypadChildProcess terminated.");
      sleep 2;
      exit(0);
   }

   while(1) {
      $button = -1;
      foreach my $col (1,2,3,4) {
         $$SensorChip{$chip}{'Obj'}->write_byte($cols{$col},
                                    $$MCP23017{ $$KeypadData{$Keypad}{'Col'} });
         sleep 0.02;         # Delay for button debounce.
         $row = $$SensorChip{$chip}{'Obj'}->read_byte($$MCP23017{
                                      $$KeypadData{$Keypad}{'Row'} }) & 0xC3;
         &DisplayDebug(3, "ReadKeypad, Keypad: $Keypad   col: " .
                        sprintf("%0.8b", $cols{$col}) . "   row: " .
                        sprintf("%0.8b", $row));

         # Process if valid single button keypress. Ignore held down button.
         if ($row == 0xC2 or $row == 0xC1 or $row == 0x43 or $row == 0x83) {
            $button = $matrix{$col}{$row};          # Get keypress result value.
            if ($button != $$KeypadData{$Keypad}{'Last'}) {
```

```
241                $$KeypadData{$Keypad}{'Last'} = $button;
242                print STDERR "$button";              # Send key press.
243                &DisplayDebug(3, "ReadKeypad, button '$button' pressed.");
244             }
245             last;
246          }
247       }
248
249       # Clear 'Last' if no button is pressed.
250       if ($button == -1 and $$KeypadData{$Keypad}{'Last'} != -1) {
251          $$KeypadData{$Keypad}{'Last'} = -1;
252          sleep 0.02;        # Delay for button debounce.
253       }
254       sleep 0.1;            # Loop delay.
255    }
256
257    &DisplayMessage("KeypadChildProcess terminated.");
258    sleep 2;
259    exit(0);
260 }
261
262 # ============================================================================
263 # FUNCTION:  ButtonChildProcess
264 #
265 # DESCRIPTION:
266 #    This routine is launched as a child process during main program startup
267 #    and is used to return user input from the 'Storm K Range' 1x4 button
268 #    keypad. This keypad is connected to a MCP23017 port as follows.
269 #
270 #       button    D   C   B   A
271 #                 |   |   |   |
272 #       common  --o---o---o---o
273 #
274 #         ButtonPad 1:    c  D C B A     ButtonPad 2:     c  D C B A
275 #         Button pin:     1  2 3 4 5     Button pin:      1  2 3 4 5
276 #         GPIOA pin:      2  6 5 4 3     GPIOA pin:       2 10 9 8 7
277 #         GPIOA bit:         0 1 2 3     GPIOA bit:          4 5 6 7
278 #
279 #    A dedicated child process is used to improve the reliability of a double
280 #    button press. Forks::Super is used between the parent and child to read
281 #    data from the child's STDERR filehandle. Do no other output to STDERR
282 #    within this routine. DisplayMessage and DisplayDebug are permitted since
283 #    they use STDOUT for messaging.
284 #
285 #    Two button data messgaes are generated, single press (s<num>) and double
286 #    press (d<num>). <num> is the button index in the %ButtonData hash. The
287 #    parent must read the child's data at a rate greater than the expected
288 #    user input rate.
289 #
290 #    Multiple button press events may be present in a message, e.g. 's01d01'.
291 #    Check first for d01 input and discard the s01 input if present.
292 #
293 #    The %ButtonData{<num>}{'Obj'} references must be set prior to launching
294 #    this child process.
295 #
296 # CALLING SYNTAX:
297 #    $ButtonChildPid = fork {os_priority => 4, sub => \&ButtonChildProcess,
298 #                            child_fh => "err socket",
299 #                            args => [ \%ButtonData, \%MCP23017, \%SensorChip ] };
300 #
```

```
#    $read_button = Forks::Super::read_stderr($ButtonChildPid);
#
# ARGUMENTS:
#    $ButtonData            Pointer to %ButtonData hash.
#    $MCP23017              Pointer to MCP23017 internal register definitions
#    $SensorChip            Pointer to %SensorChip hash.
#
# RETURNED VALUES:
#    s<num> - Button <num> has been single pressed.
#    d<num> - Button <num> has been double pressed.
#
# ACCESSED GLOBAL VARIABLES:
#    $main::ChildName
# ============================================================================
sub ButtonChildProcess {
   my($ButtonData, $MCP23017, $SensorChip) = @_;
   my($port, $mask, $chip, $check);

   $main::ChildName = 'ButtonChildProcess';
   &DisplayMessage("ButtonChildProcess started.");

   while(1) {
      foreach my $button (sort keys %$ButtonData) {
         next if ($button == 0xFF);             # Ignore shutdown button entry
         $chip = $$ButtonData{$button}{'Chip'};
         if ($$ButtonData{$button}{'Bit'} =~ m/^(GPIO.)(\d)/) {
            $port = $1;
            $mask = 1 << $2;

            # Read the port and isolate the bit value.
            $check = $$SensorChip{$chip}{'Obj'}->read_byte($$MCP23017{$port});
            $check = $check & $mask;
            # 'Last' is used to handle a held down button. Only use the
            # transition from 0 to 1 as a button press.
            if ($check != 0) {
               if ($$ButtonData{$button}{'Last'} == 1) {
                  $$ButtonData{$button}{'PressTime'} = gettimeofday;
                  next;
               }
               if ((gettimeofday - $$ButtonData{$button}{'PressTime'}) < 1) {
                  print STDERR "d${button}";               # Send double press.
                  $$ButtonData{$button}{'PressTime'} = 0;  # New press cycle.
                  &DisplayDebug(1, "ButtonChildProcess, button: d${button}");
               }
               else {
                  print STDERR "s${button}";               # Send single press
                  $$ButtonData{$button}{'PressTime'} = gettimeofday;
                  &DisplayDebug(1, "ButtonChildProcess, button: s${button}");
               }
               $$ButtonData{$button}{'Last'} = 1;
            }
            else {
               $$ButtonData{$button}{'Last'} = 0;          # Button released.
            }
         }
      }
      sleep 0.05;             # Loop delay.
   }

   &DisplayMessage("ButtonChildProcess terminated.");
```

```
361     sleep 2;
362     exit(0);
363  }
364
365  # ============================================================================
366  # FUNCTION:  PositionChildProcess
367  #
368  # DESCRIPTION:
369  #    This routine is launched as a child process during main program startup.
370  #    It periodically reads the train hold position sensors associated with the
371  #    holdover tracks and sets the appropriate panel LEDs to provide a visual
372  #    indication of train position in these hidden tracks. Warning point (yellow)
373  #    and stop point (red) LEDs are used.
374  #
375  # CALLING SYNTAX:
376  #    $result = &PositionChildProcess(\%SensorBit, \%PositionLed, \%SensorChip,
377  #                                    \%MCP23017);
378  #
379  # ARGUMENTS:
380  #    $SensorBit          Pointer to %SensorBit hash.
381  #    $PositionLed        Pointer to %PositionLed hash.
382  #    $SensorChip         Pointer to %SensorChip hash.
383  #    $MCP23017           Pointer to MCP23017 internal register definitions.
384  #
385  # RETURNED VALUES:
386  #    0 = Success,  1 = Error.
387  #
388  # ACCESSED GLOBAL VARIABLES:
389  #    $main::ChildName
390  # ============================================================================
391  sub PositionChildProcess {
392     my($SensorBit, $PositionLed, $SensorChip, $MCP23017) = @_;
393     my($chip, $port, $pos, $senBit, $ledBits);
394
395     $main::ChildName = 'PositionChildProcess';
396     &DisplayMessage("PositionChildProcess started.");
397
398     while(1) {
399        foreach my $led (sort keys(%$PositionLed)) {
400           $chip = $$SensorBit{$led}{'Chip'};
401           if ($$SensorBit{$led}{'Bit'} =~ m/^(GPIO.)(\d)/) {
402              $port = $1;
403              $pos = $2;
404
405              # Read sensor port and isolate the bit value.
406              $senBit = $$SensorChip{$chip}{'Obj'}->read_byte($$MCP23017{$port});
407              $senBit = ($senBit >> $pos) & 1;   # Position and isolate.
408
409              $chip = $$PositionLed{$led}{'Chip'};
410              if ($$PositionLed{$led}{'Bit'} =~ m/^(GPIO.)(\d)/) {
411                 $port = $1;
412                 $pos = $2;
413
414                 # Update associated LED bit value.
415                 $ledBits = $$SensorChip{$chip}{'Obj'}->read_byte(
416                            $$MCP23017{$port});
417                 $ledBits = $ledBits & (~(1 << $pos));     # Clear bit position.
418                 $ledBits = $ledBits | ($senBit << $pos);  # Set bit position.
419                 $$SensorChip{$chip}{'Obj'}->write_byte($ledBits,
420                                       $$MCP23017{ $$PositionLed{$led}{'Olat'} });
```

```
421                 }
422             }
423         }
424         sleep 0.5;              # Loop delay.
425     }
426
427     &DisplayMessage("PositionChildProcess terminated.");
428     sleep 1;
429     exit(0);
430 }
431
432 # ============================================================================
433 # FUNCTION:  GetSensorBit
434 #
435 # DESCRIPTION:
436 #    This routine returns the current value of the specified sensor bit. The
437 #    proper SensorState hash index is determined based on the requested bit
438 #    number. The bit number must include leading zero (0) if less that 10 for
439 #    proper index key in %SensorBit hash.
440 #
441 # CALLING SYNTAX:
442 #    $result = &GetSensorBit($BitNumber, \%SensorBit, \%SensorState);
443 #
444 # ARGUMENTS:
445 #    $BitNumber             Bit position to check (index in %SensorBit)
446 #    $SensorBit             Pointer to %SensorBit hash.
447 #    $SensorState           Pointer to %SensorState hash.
448 #
449 # RETURNED VALUES:
450 #    Bit value: 0 or 1
451 #
452 # ACCESSED GLOBAL VARIABLES:
453 #    None.
454 # ============================================================================
455 sub GetSensorBit {
456     my($BitNumber, $SensorBit, $SensorState) = @_;
457     my($bitMask) = 1 << ($BitNumber % 16);
458
459     return 1 if ($$SensorState{ $$SensorBit{$BitNumber}{'Chip'} } & $bitMask);
460
461     return 0;
462 }
463
464 # ============================================================================
465 # FUNCTION:  TestSensorBits
466 #
467 # DESCRIPTION:
468 #    This routine displays the sensor state bits on the console. The user can
469 #    manually activate each sensor and observe the expected result. This test
470 #    loops indefinitely until the user enters ctrl+c.
471 #
472 # CALLING SYNTAX:
473 #    $result = &TestSensorBits($Range, \%MCP23017, \%SensorChip, \%SensorState);
474 #
475 # ARGUMENTS:
476 #    $Range                   Chip number or range to use.
477 #    $MCP23017                Pointer to MCP23017 internal register definitions
478 #    $SensorChip              Pointer to %SensorChip hash.
479 #    $SensorState             Pointer to %SensorState hash.
480 #
```

```perl
# RETURNED VALUES:
#    0 = Success,  1 = Error.
#
# ACCESSED GLOBAL VARIABLES:
#    $main::MainRun
# ============================================================================
sub TestSensorBits {
   my($Range, $MCP23017, $SensorChip, $SensorState) = @_;
   my($chip, $start, $end, $msg, @chipList);
   my($cntr) = 0;

   &DisplayDebug(2, "TestSensorBits, Entry ... Range: '$Range'");

   if ($Range =~ m/(\d+):(\d+)/) {   # Range specified.
      $start = $1;
      $end = $2;
      if ($start > $end or $start < 1 or $start > 4 or $end < 1 or $end > 4) {
         &DisplayError("TestSensorBits, invalid sensor chip range: '$Range'");
         return 1;
      }
      for ($chip = $start; $chip <= $end; $chip++) {
         push (@chipList, $chip);
      }
   }
   else {
      @chipList = split(",", $Range);
   }
   &DisplayDebug(1, "TestSensorBits, chipList: '@chipList'");

   &DisplayMessage("TestSensorBits - enter ctrl+c to exit.\n");
   &DisplayMessage("TestSensorBits -----                                " .
                   "bit 76543210      bit 76543210");
   while($main::MainRun) {
      foreach my $chip (@chipList) {
         $msg = sprintf("%0.6d", $cntr);     # Show program activity on console.
         if (exists($$SensorChip{$chip})) {
            $$SensorState{$chip} =
               ($$SensorChip{$chip}{'Obj'}->read_byte($$MCP23017{'GPIOB'}) << 8) |
                $$SensorChip{$chip}{'Obj'}->read_byte($$MCP23017{'GPIOA'});

            $msg = $msg . " I2C Address: " .
                          sprintf("0x%.2X", $$SensorChip{$chip}{'Addr'});
            $msg = $msg . "   GPIOB: " .
                          sprintf("%0.8b", ($$SensorState{$chip} >> 8));
            $msg = $msg . "   GPIOA: " .
                          sprintf("%0.8b", ($$SensorState{$chip} & 0xFF));
            &DisplayMessage("TestSensorBits - $msg");
         }
         else {
            &DisplayError("TestSensorBits, invalid sensor range: '$Range'");
            return 1;
         }
      }
      $cntr++;
      sleep 1;
   }
   return 0;
}

# ============================================================================
```

```perl
# FUNCTION:  TestSensorTones
#
# DESCRIPTION:
#    This routine tests all sensors. A sensor ID number of tones are sounded
#    when a sensor becomes active and a double tone is sounded when a sensor
#    becomes inactive. This facilitates operability testing of the layout
#    remote sensors; e.g. by manually blocking the IR light path. This test
#    loops indefinitely until the user enters ctrl+c.
#
# CALLING SYNTAX:
#    $result = &TestSensorTones(\%MCP23017, \%SensorChip, \%SensorState,
#                               \%SensorBit);
#
# ARGUMENTS:
#    $MCP23017             Pointer to MCP23017 internal register definitions
#    $SensorChip           Pointer to %SensorChip hash.
#    $SensorState          Pointer to %SensorState hash.
#    $SensorBit            Pointer to %SensorBit hash.
#
# RETURNED VALUES:
#    0 = Success,  1 = Error.
#
# ACCESSED GLOBAL VARIABLES:
#    $main::MainRun
# ============================================================================
sub TestSensorTones {
   my($MCP23017, $SensorChip, $SensorState, $SensorBit) = @_;
   my($bitNmbr, $cur, $prev, $x, $y);
   my(%localSensorState) = ('1' => 0, '2' => 0);

   &DisplayDebug(2, "TestSensorTones, Entry ...");

   &DisplayMessage("TestSensorTones, ============================");
   &DisplayMessage("TestSensorTones, Waiting for change in sensor " .
                   "bits 0-31 ...");
   while($main::MainRun) {

      # Get all sensor states.
      $localSensorState{'1'} =
      ($$SensorChip{'1'}{'Obj'}->read_byte($$MCP23017{'GPIOB'}) << 8) |
       $$SensorChip{'1'}{'Obj'}->read_byte($$MCP23017{'GPIOA'});
      $localSensorState{'2'} =
      ($$SensorChip{'2'}{'Obj'}->read_byte($$MCP23017{'GPIOB'}) << 8) |
       $$SensorChip{'2'}{'Obj'}->read_byte($$MCP23017{'GPIOA'});

      # If a bit has changed, report change.
      for ($x = 0; $x < 32; $x++) {
         $bitNmbr = sprintf("%0.2d",$x);
         next if ($$SensorBit{$bitNmbr}{'Desc'} =~ m/spare/ or
                  $$SensorBit{$bitNmbr}{'Desc'} =~ m/Unused/ );
         $cur  = &GetSensorBit($bitNmbr, $SensorBit, \%localSensorState);
         $prev = &GetSensorBit($bitNmbr, $SensorBit, $SensorState);

         if (($cur - $prev) == 1) {      # Bit now set.
            &DisplayMessage("TestSensorTones, Sensor bit $bitNmbr" .
                            " has set   (1). [" .
                            $$SensorBit{$bitNmbr}{'Desc'} . "]");
            for ($y = 0; $y < $x; $y++) {
               &PlaySound("Lock.wav");
               sleep 0.5;
```

```
            }
            last;                       # Skip remaining bits
         }
         elsif (($prev - $cur) == 1) {   # Bit now reset.
            &DisplayMessage("TestSensorTones, Sensor bit $bitNmbr" .
                            " has reset (0). [" .
                            $$SensorBit{$bitNmbr}{'Desc'} . "]");
            &PlaySound("Unlock.wav");
            last;                       # Skip remaining bits
         }
      }

      # Update %SensorState hash with just read sensor states.
      $$SensorState{'1'} = $localSensorState{'1'};
      $$SensorState{'2'} = $localSensorState{'2'};
      sleep 0.5;
   }
   return 0;
}

# ============================================================================
# FUNCTION:  TestKeypad
#
# DESCRIPTION:
#    This routine displays the pressed button on the keypad. It also sets and
#    resets the 1st entry LED with each key press. The individual turnout
#    buttons will also be displayed when pressed. This test loops until the
#    user enters ctrl+c.
#
# CALLING SYNTAX:
#    $result = &TestKeypad($Keypad, \%KeypadData, \%ButtonData, \%GpioData,
#                          \%MCP23017, \%SensorChip, \$KeypadChildPid,
#                          \$ButtonChildPid);
#
# ARGUMENTS:
#    $KeypadId              KeypadData entry to test.
#    $KeypadData            Pointer to %KeypadData hash.
#    $ButtonData            Pointer to %ButtonData hash.
#    $GpioData              Pointer to %GpioData hash.
#    $MCP23017              Pointer to MCP23017 internal register definitions
#    $SensorChip            Pointer to %SensorChip hash.
#    $KeypadChildPid        Pointer to KeypadChild pid value.
#    $ButtonChildPid        Pointer to ButtonChild pid value.
#
# RETURNED VALUES:
#    0 = Success,  1 = Error.
#
# ACCESSED GLOBAL VARIABLES:
#    $main::MainRun
# ============================================================================
sub TestKeypad {
   my($KeypadId, $KeypadData, $ButtonData, $GpioData, $MCP23017, $SensorChip,
      $KeypadChildPid, $ButtonChildPid) = @_;
   my($button, $value, $lockLed, $key, @input);

   &DisplayDebug(2, "TestKeypad, Entry ... KeypadId: $KeypadId   KeypadChildPid: " .
                 "$$KeypadChildPid   ButtonChildPid: $$ButtonChildPid");

   $KeypadId = sprintf("%0.2d",$KeypadId);   # Add leading 0 for proper key.
   if (exists $$KeypadData{$KeypadId}) {
```

```
661         while($main::MainRun) {
662
663            # Keypad buttons.
664            $button = Forks::Super::read_stderr($$KeypadChildPid);
665            if ($button eq '') {
666               &DisplayMessage("TestKeypad, KeypadId: $KeypadId - No " .
667                               "button pressed.");
668            }
669            else {
670               $button = substr($button, 0, 1);  # 1st character only if multiple.
671               &DisplayMessage("TestKeypad, KeypadId: $KeypadId - " .
672                               "keypad button pressed: '$button'");
673
674               # Read keypad 1st entry LED.
675               $value = $$GpioData{ $$KeypadData{$KeypadId}{'Gpio'} }{'Obj'}->read;
676
677               # Compliment the value.
678               $value = (~$value) & 1;
679
680               # Write keypad 1st entry LED.
681               $$GpioData{ $$KeypadData{$KeypadId}{'Gpio'} }{'Obj'}->write($value);
682               &DisplayMessage("TestKeypad, KeypadId: $KeypadId - 1st " .
683                               "entry LED set to $value");
684            }
685
686            # Single buttons.
687            $button = Forks::Super::read_stderr($$ButtonChildPid);
688            if ($button ne '') {
689               if ($button =~ m/d(\d+)/) {
690                  $value = $1;
691                  &DisplayMessage("TestKeypad, button double press: " .
692                                  "$$ButtonData{$1}{'Desc'}");
693               }
694               elsif ($button =~ m/s(\d+)/) {
695                  $value = $1;
696                  &DisplayMessage("TestKeypad, button single press: " .
697                                  "$$ButtonData{$1}{'Desc'}");
698               }
699               else {
700                  &DisplayMessage("TestKeypad, invalid button response: " .
701                                  "'$button'");
702               }
703               if ($value ge '04' and $value le '07') {
704
705                  # Read Holdover route lock LED.
706                  $lockLed = $$GpioData{'GPIO26_HLCK'}{'Obj'}->read;
707
708                  # Compliment the value.
709                  $lockLed = (~$lockLed) & 1;
710
711                  # Write keypad 1st entry LED.
712                  $$GpioData{'GPIO26_HLCK'}{'Obj'}->write($lockLed);
713                  &DisplayMessage("TestKeypad, Lock led set to $lockLed");
714               }
715            }
716            else {
717               &DisplayMessage("TestKeypad, No single button input.");
718            }
719
720            # Read and display shutdown button state.
```

```
721             $button = $$GpioData{'GPIO21_SHDN'}{'Obj'}->read;
722             &DisplayMessage("TestKeypad, Shutdown button: $button");
723             sleep 2;
724          }
725       }
726       else {
727          &DisplayError("TestKeypad, Keypad $KeypadId is not supported.");
728          return 1;
729       }
730       return 0;
731    }
732
733    return 1;
734
```