Application Report
SLAA730A – February 2017 – Revised August 2017

PGA460 Software Development Guide

Akeem Whitehead

ABSTRACT

The objective of this guide is to explain the high-level software flow for platform development with the PGA460 device using UART, Time Command Interface (TCI), and/or One-Wire UART (OWU) communication.

This report contains a software development flow-chart, one-time mass-production initialization requirements, main executable routines, and source code examples. The example software is written in Energia for the MSP-EXP430F5529LP, but can be adapted to any TI LaunchPad™ development kit.

Contents

List of Figures

Trademarks

LaunchPad is a trademark of Texas Instruments.
All other trademarks are the property of their respective owners.

1 Introduction to High-Level Software Flow

The PGA460 device can only be operated as a slave device and must be paired with an external microcontroller unit (MCU) which acts as the master device. The master device is responsible for the initialization, configuration, and regular polling operation of the PGA460 device. Figure 1 shows the high-level overview of the software flow for standard PGA460 operation. There are three main components of the software flow:

1. The main file
2. The PGA460 header and driver files
3. The master controller header and driver files

After system initialization, the program of the main file loops the routine shown in Figure 1 for reading the ultrasonic time-of-flight results from the PGA460 device. For the purpose of this guide, the PGA460-specific code is referenced from the PGA460_USSC Energia IDE library file containing the *pga460_ussc.cpp* and *pga460_ussc.h* files. The master controller used in this example is the MSP430F5529 MCU on the MSP-EXP430F5529LP platform. The PGA460-Q1 EVM hardware is used to demonstrate the operation of the software.

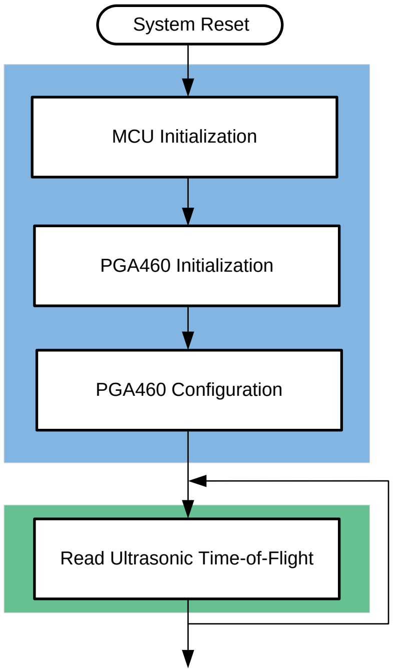


Figure 1. High-Level Software Flowchart for PGA460 Operation

2 Main File

The main file hosts the high-level code and routines required to initialize the master controller and PGA460 device, and run PGA460-specific operations. Figure 2 shows the software flowchart sequence and the mapping of device specific functions.

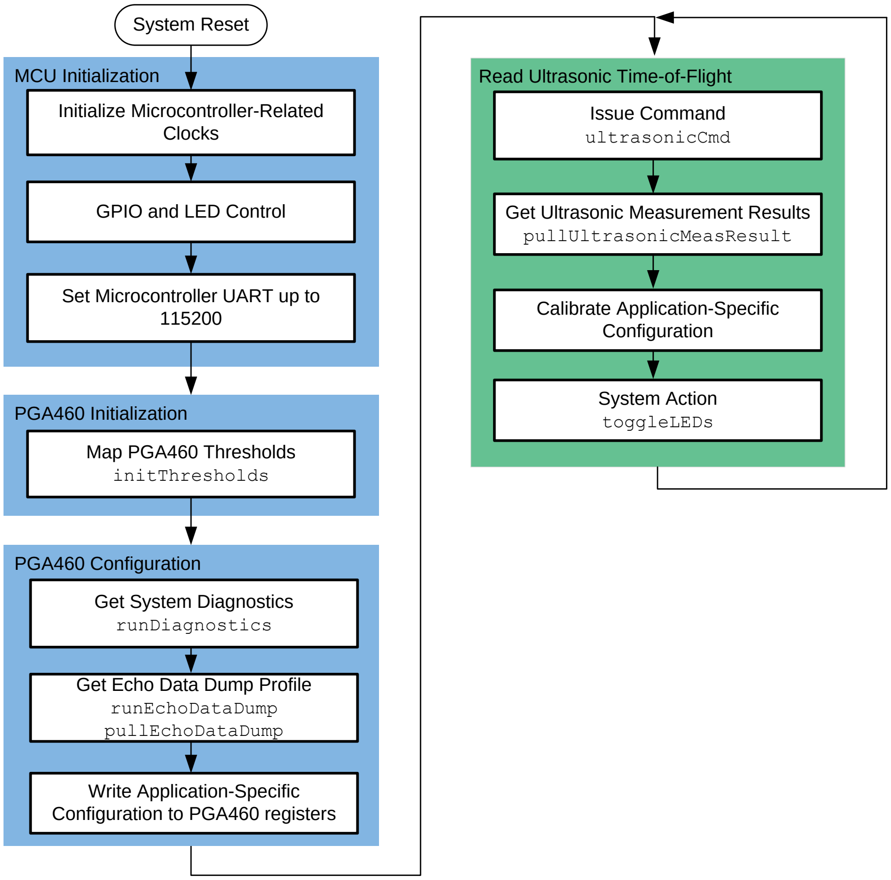

System Reset
MCU Initialization
Initialize Microcontroller-Related Clocks
GPIO and LED Control
Set Microcontroller UART up to 115200
PGA460 Initialization
Map PGA460 Thresholds
initThresholds
PGA460 Configuration
Get System Diagnostics
runDiagnostics
Get Echo Data Dump Profile
runEchoDataDump
pullEchoDataDump
Write Application-Specific Configuration to PGA460 registers
Read Ultrasonic Time-of-Flight
Issue Command
ultrasonicCmd
Get Ultrasonic Measurement Results
pullUltrasonicMeasResult
Calibrate Application-Specific Configuration
System Action
toggleLEDs

Figure 2. Detailed Software Sequence Flowchart

2.1 MCU Initialization

The internal clocks, UART ports, and GPIOs of the master controller must be configured prior to initialization of the PGA460 registers.

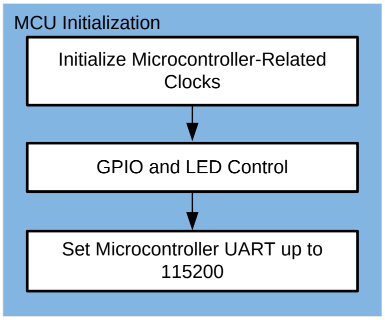

MCU Initialization
Initialize Microcontroller-Related Clocks
GPIO and LED Control
Set Microcontroller UART up to 115200

Figure 3. MCU Initialization

The UART terminals of the master controller must be adapted to the PGA460-compatible format and baud rate. The PGA460 UART supports up to 115.2 kBaud, and is always formatted in 8 data bits, two stop bits, no parity, and no flow control. Ensure the master controller clock is referenced to a source and frequency that can support the designated baud rate.

If the UART logic level of the PGA460 device is to be dynamically adjusted, then a GPIO output is required to drive the PGA460 SCLK pin high or low at system startup to configure the logic level to 5 V or 3.3 V respectively. Because the logic level is typically limited to a single voltage, and fixed for most systems, the default logic level is referenced to 3.3 V when the SCLK pin is floating, or pulled-low through a fixed resistor as on the PGA460-Q1 EVM. TI does not recommend leaving the SCLK pin floating.

One-Wire UART uses the same master port configuration as the standard two-wire UART, assuming the system implements a One-Wire physical interface. This physical interface acts as a serial transceiver to convert the independent logic-level Tx and Rx pins to a single-wire bidirectional bus for connection to the PGA460-Q1 IO pin. The Texas Instrument SN65HVDA100-Q1 is an example device that can be used for PGA460-Q1 OWU mode up to 19.2 kbps.

In TCI mode, UART port initialization is not required. Instead, two GPIOs can be repurposed to bit-banging the TCI transmit signal, and use a timer, logic-level transient detector, or interrupt to monitor and decode the TCI receive signal. Similarly to the OWU hardware implementation, a One-Wire physical interface is required to convert the independent logic-level GPO and GPI pins to a single-wire bidirectional signal for connection to the PGA460-Q1 IO pin. Because the TCI baud rate is an equivalent of 10-kHz, the switching requirements are not as stringent as OWU. An alternative implementation to achieve TCI communication is to use the MOSI and MISO pins of the SPI port (the SCLK and CS pins can remain floating). In this SPI-to-TCI conversion, a single TCI bit of 300 µs can be implemented as 3 bytes of SPI transmit data at 80-kHz. At the receiver, the SPI read would record incoming TCI data at the same speed of 80 kHz. Although the SPI implementation requires more memory than a GPIO bit-bang, or transient, approach, this continuous sampling method ensures no logic transients are skipped.

The PGA460-Q1 EVM uses three LEDs to visually indicate ranging performance or update the diagnostic status of the module. Although the LEDs are optional, the master controller is typically configured to respond to the ultrasonic time-of-flight data by toggling an electromechanical switch, activating a subsequent system module, or displaying information in response. Configure the external functions of the master controller as required by the host system.

2.2 PGA460 Initialization

Initialization of the PGA460 device only requires a write to the threshold registers, but understanding how to implement the checksum and optional CRC algorithm is required to perform any read or write command.

2.2.1 PGA460 Checksum Calculation

The frame checksum value is required and used to ensure that data communicated between the master controller and the PGA460 device has not been compromised or corrupted when transmitted or received. The frame checksum value is generated by both the master and slave devices, and is added after the data field. The checksum is calculated as the inverted eight bit sum with carry-over on all bits in the frame.

In TCI mode, the checksum calculation occurs bytewise, starting from the most-significant bit (MSB). The MSB is the read-write (R/W) bit in the PGA460 write operation. For the PGA460 read operation, the MSB starts in the data field. In cases where the number of bits on which the checksum field is calculated yields a non-multiple of eight, the checksum operation pads trailing zeros until the closest multiple of eight is achieved.

In UART mode, a checksum field is transmitted as the last field of every frame. The checksum contains the value of the inverted byte sum with carry operation over all data fields and the command field (command field for master only). On a master-to-PGA460 transmission, the checksum field is calculated by the master device and checked by the PGA460 device. On a PGA460-to-master transmission, the PGA460 device generates the checksum and the master has the option to validate the data integrity. The format of the checksum is identical to the data field and procedure for calculating the checksum in TCI mode. Because the UART interface is a byte-based interface, no zero padding occurs in the process of calculating the checksum.

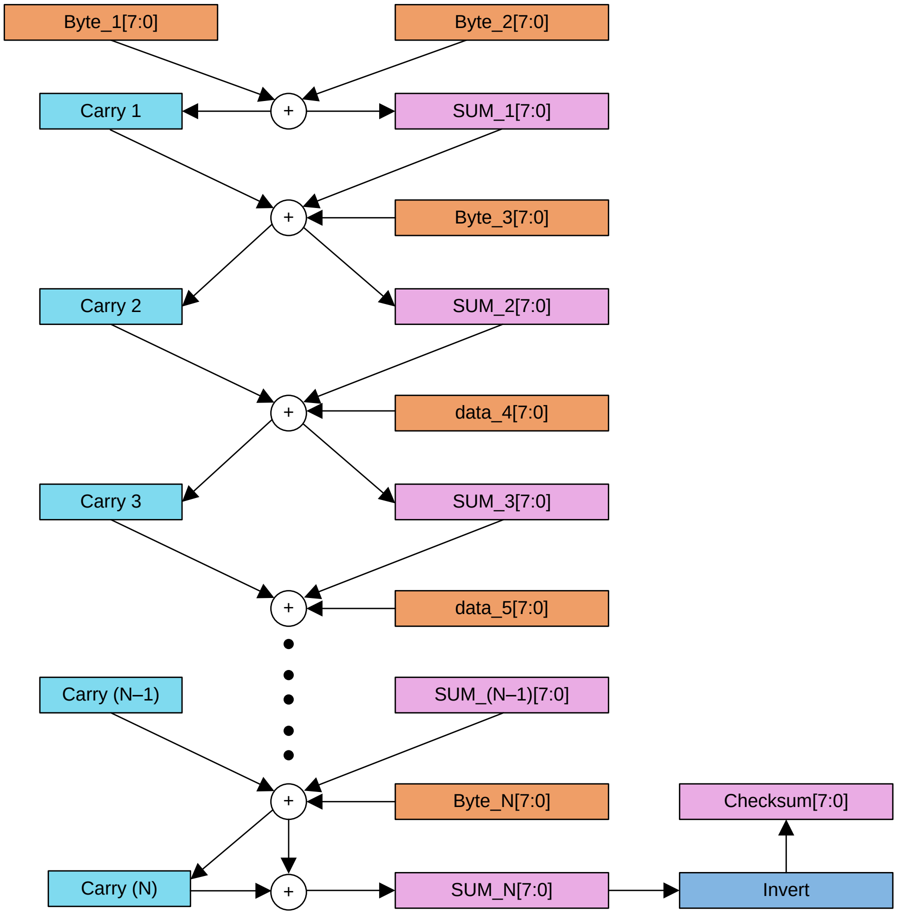


Figure 4. Checksum Calculation Flow Chart

2.2.2 PGA460 Threshold Initialization

The PGA460 threshold registers are stored in volatile memory; therefore, the threshold values are random upon start-up.

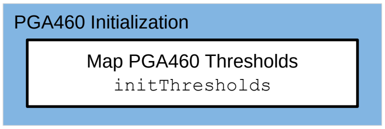


Figure 5. PGA460 Initialization

The PGA460 device only requires that the threshold registers be written to at least once upon start-up to enable the BURST/LISTEN command. The hard requirement to clear the associated threshold CRC is for at least one bit from only a single threshold register (either Preset 1 or Preset 2) to be updated to enable the driver block. Even if the ultrasonic measurement result command is not used to compare the threshold to the echo data dump profile, the threshold CRC must be cleared to allow any transducer excitation.

If only one preset is used, then updating only the threshold of the designated preset is recommended to reduce PGA460 initialization time.

2.2.3 First Time Bulk EEPROM Write and Burn

If the PGA460 is pristine, or has never been loaded with the optimized register settings, a one-time EEPROM bulk write and EEPROM burn is required. This routine ensures that the default drive, receive, and diagnostic settings loaded from EEPROM match the system expectations rather than the TI default factory values. First, load the user EEPROM values and then program the EEPROM.

To program the EEPROM, follow these steps:

Step 1. Send an EEPROM program command using UART or TCI with a unique unlock pattern of 4-bits. The program bit is set to 0 in the 0x40 register. The unlock passcode is 0xDh.

Step 2. Immediately send the same UART or TCI command with the program bit set to 1.

If any other command is issued after the unlock code (Step 1), the EEPROM program sequence is aborted. If the unlock command in Step 1 is not correct, the EEPROM is not programmed. The EEPROM is locked again automatically after each program command.

2.2.4 Memory CRC Calculation (Optional)

The PGA460-Q1 implements a cyclic redundancy check (CRC) that is a self-contained algorithm to verify the integrity of the EEPROM stored data and threshold settings. When an EEPROM program or EEPROM-reload operation is executed, or when a threshold register is written, the CRC controller calculates the correct CRC value and writes it to the corresponding registers. For EEPROM memory, this value is written to the EE_CRC register. For threshold settings, this value is written to the THR_CRC register. A CRC is performed at power-up when an EEPROM reload command is issued. The CRC algorithm for all memory blocks is the same, with an initial seed value of 0xFF, and uses MSB ordering.

$$\text{CRC algorithm} = X^8 + X^2 + X + 1 \text{ (ATM HEC)} \tag{1}$$

This calculation is performed bytewise starting from the MSB to the LSB. The data is concatenated as follows:

- For EEPROM memory: Concatenation starts with MSB USER_DATA1 (0x00) to LSB P2_GAIN_CTRL (0x2A) and calculated CRC is stored in the EE_CRC register (0x2B)
- For threshold settings: Concatenation starts with MSB P1_THR_0 (0x5F) to LSB P2_THR_15 (0x7E) and calculated CRC is stored in the THR_CRC register (0x7F)

NOTE: The master controller is not required to implement or compute the CRC algorithm. The PGA460 device automatically and internally performs all CRC calculations and checks. This is an optional function the master can implement for redundancy.

The results of the CRC check are stored in the DEV_STAT0 register and can be read directly through the UART interface, while the time-command interface reports these in the STAT 1 and STAT 3 status bits.

2.3 PGA460 Configuration

After the thresholds have been updated, the system can now continuously run the burst-and-listen command to pull resulting measurement data. However, for proper calibration, the system diagnostics and echo data dump can be run prior to continuous operation to determine if the ultrasonic module is behaving as expected. By comparing the diagnostic results and data dump output to a nominal profile, the PGA460 device settings can be fine-tuned further for optimal performance.

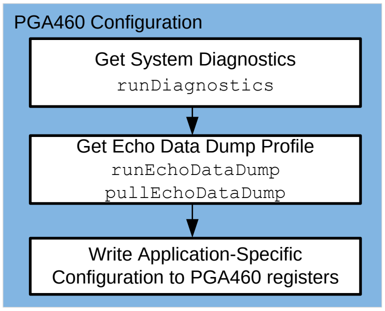


Figure 6. PGA460 Configuration

2.3.1 System Diagnostics

The PGA460 device offers a system level diagnostic of the resonant frequency of the transducer, decay period of the resonant energy immediately after bursting, excitation voltage of the transducer, ambient temperature, and a noise-floor level measurement. All of these elements can be checked when running the example system diagnostic function. The frequency, decay, and voltage measurements are useful for ensuring the driver component and transducer are not damaged or unintentionally loaded. The noise-floor measurement can be used to increase and remap the threshold profile for improved margin, and set the digital signal processor to filter more aggressively in the event more noise is present. The temperature measurement can be used to account for the change in the speed of sound across temperature to correctly calculate distance and apply any external passives for tuning.

2.3.2 Echo Data Dump

The echo data dump is typically used during the development and debugging stages of the PGA460 device when initially optimizing the device settings and threshold for the best case signal-to-noise ratio (SNR) scaling. Although not required for normal operation, the echo data dump provides more detailed data when compared to the ultrasonic measurement results to determine if the overall ultrasonic profile appears as expected. The echo data dump can also be used for advanced functions carried out by the master controller, such as averaging and automatically mapping a threshold for a no-object profile.

2.4 Read Ultrasonic Time-of-Flight

The PGA460 device spends the majority of the time looping the BURST/LISTEN command. Users have the option of using either one or both presets. To achieve the best-case minimum and maximum ranging performance, Preset 1 is typically optimized for short ranges, while Preset 2 is optimized for long ranges. The definitions of a short and long range or distance is subjective, but for this discussion, assume that a short distance is within 1 m and a long distance is beyond 1 m. Optimizing a single preset for short and long distance is possible with some losses to the best-case minimum and maximum values.

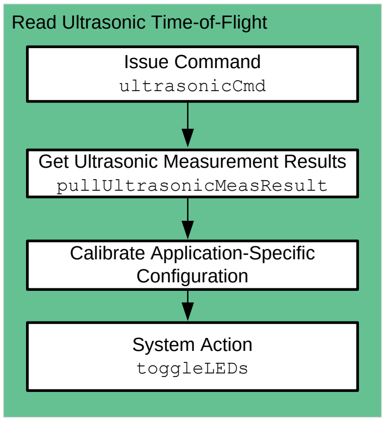

Read Ultrasonic Time-of-Flight
Issue Command
ultrasonicCmd
Get Ultrasonic Measurement Results
pullUltrasonicMeasResult
Calibrate Application-Specific Configuration
System Action
toggleLEDs

Figure 7. Read Ultrasonic Time-of-Flight

2.4.1 Ultrasonic Measurement Results

The PGA460 device captures the interrupt time and outputs the distance equivalent, width, and peak amplitude for the returning echoes when the threshold is intersected. To solve for the time-of-flight, use velocity = distance / time. Because the speed of sound is typically at a value of 343 m/s at room temperature, and the PGA460 device outputs the round-trip time at which the threshold is intersected in 1-µs resolution after bursting, the distance to the object is computed as the product of velocity and one-way time. Use Equation 2 as the PGA460-specific equation to solve for distance in meters.

$$\text{distance (m)} = \left[\left(\frac{343\ \text{m/s}}{2}\right) \times \left(\left[\left(\text{objMSB[1]} \ll 8\right) + \text{objLSB[2]}\right] \times 0.000001\right)\right] + \left[\frac{343\ \text{m/s}}{2} \times \left(\text{Pulses} \times \left[\frac{1}{\text{Frequency}}\right]\right)\right] \quad (2)$$

2.4.2 System Action

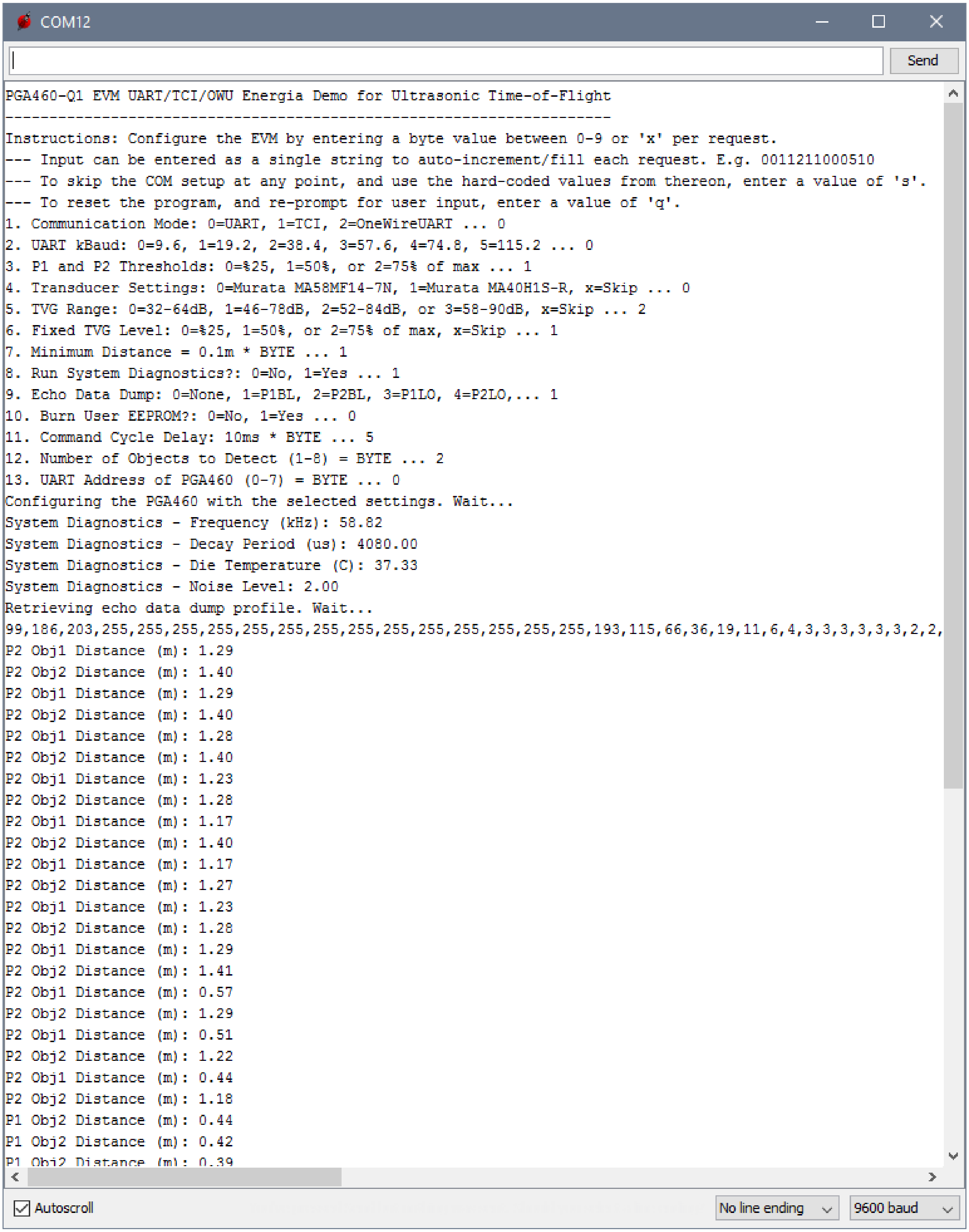
In response to distance, amplitude, width, or a combination of information, the PGA460-Q1 EVM toggles the on-board LEDs to indicate the distance to the targeted object as short, mid, or long range. The more LEDs illuminated, the further the object from the transducer. This feature allows the EVM to be run in a basic standalone mode without the need for a computer or COM terminal.

3 Energia Example

The example PGA460 program is written for the Energia IDE, and is intended for cross-platform LaunchPad evaluation. The *GetDistance.ino* project executes the previously mentioned time-of-flight to distance command loop in either COM terminal mode or standalone mode based on the comment status of *#define userInputMode*.

3.1 COM Terminal Mode

The code prompts the user to input the system operating conditions when using the PGA460-Q1 EVM. After the user has entered the operating conditions through the COM terminal, the code optionally executes all applicable requests, including diagnostics, echo data dump output, and EEPROM programming, followed by a continuously looped burst-and-listen command with interim ultrasonic measurement result data pulled. To stop the BURST/LISTEN looping, type a value of *q* in the COM terminal. This entry resets the code and prompts for the user input values.



```
PGA460-Q1 EVM UART/TCI/OWU Energia Demo for Ultrasonic Time-of-Flight
-------------------------------------------------------------------------
Instructions: Configure the EVM by entering a byte value between 0-9 or 'x' per request.
--- Input can be entered as a single string to auto-increment/fill each request. E.g. 0011211000510
--- To skip the COM setup at any point, and use the hard-coded values from thereon, enter a value of 's'.
--- To reset the program, and re-prompt for user input, enter a value of 'q'.
1. Communication Mode: 0=UART, 1=TCI, 2=OneWireUART ... 0
2. UART kBaud: 0=9.6, 1=19.2, 2=38.4, 3=57.6, 4=74.8, 5=115.2 ... 0
3. P1 and P2 Thresholds: 0=%25, 1=50%, or 2=75% of max ... 1
4. Transducer Settings: 0=Murata MA58MF14-7N, 1=Murata MA40H1S-R, x=Skip ... 0
5. TVG Range: 0=32-64dB, 1=46-78dB, 2=52-84dB, or 3=58-90dB, x=Skip ... 2
6. Fixed TVG Level: 0=%25, 1=50%, or 2=75% of max, x=Skip ... 1
7. Minimum Distance = 0.1m * BYTE ... 1
8. Run System Diagnostics?: 0=No, 1=Yes ... 1
9. Echo Data Dump: 0=None, 1=P1BL, 2=P2BL, 3=P1LO, 4=P2LO,... 1
10. Burn User EEPROM?: 0=No, 1=Yes ... 0
11. Command Cycle Delay: 10ms * BYTE ... 5
12. Number of Objects to Detect (1-8) = BYTE ... 2
13. UART Address of PGA460 (0-7) = BYTE ... 0
Configuring the PGA460 with the selected settings. Wait...
System Diagnostics - Frequency (kHz): 58.82
System Diagnostics - Decay Period (us): 4080.00
System Diagnostics - Die Temperature (C): 37.33
System Diagnostics - Noise Level: 2.00
Retrieving echo data dump profile. Wait...
99,186,203,255,255,255,255,255,255,255,255,255,255,255,255,255,255,193,115,66,36,19,11,6,4,3,3,3,3,3,3,2,2,
P2 Obj1 Distance (m): 1.29
P2 Obj2 Distance (m): 1.40
P2 Obj1 Distance (m): 1.29
P2 Obj2 Distance (m): 1.40
P2 Obj1 Distance (m): 1.28
P2 Obj2 Distance (m): 1.40
P2 Obj1 Distance (m): 1.23
P2 Obj2 Distance (m): 1.28
P2 Obj1 Distance (m): 1.17
P2 Obj2 Distance (m): 1.40
P2 Obj1 Distance (m): 1.17
P2 Obj2 Distance (m): 1.27
P2 Obj1 Distance (m): 1.23
P2 Obj2 Distance (m): 1.28
P2 Obj1 Distance (m): 1.29
P2 Obj2 Distance (m): 1.41
P2 Obj1 Distance (m): 0.57
P2 Obj2 Distance (m): 1.29
P2 Obj1 Distance (m): 0.51
P2 Obj2 Distance (m): 1.22
P2 Obj1 Distance (m): 0.44
P2 Obj2 Distance (m): 1.18
P1 Obj2 Distance (m): 0.44
P1 Obj2 Distance (m): 0.42
P1 Obj2 Distance (m): 0.39
```

Figure 8. PGA460_USSC GetDistance—COM Terminal Mode

3.2 Standalone Mode

For a demonstration that strictly uses the BOOSTXL-PGA460 on-board LEDs, *#define userInputMode* can be commented out such that the PGA460 device is immediately configured with the hard-coded settings initialized in the script, and starts the BURST/LISTEN command looping without any prompts and user input. The PGA460 settings configured in standalone mode include the user EEPROM settings specific to the transducer, the time-varying gain profile, and the threshold mapping. The COM terminal continues to be updated in the background with the distance information in the event the user must confirm the exact values measured. The system diagnostics, echo data dump, and EEPROM burn functions are not executed in standalone mode.

4 Energia Example – GetDistance.ino

The *GetDistance.ino* code is as follows:

```
/*------------------------------------------------- GetDistance -----
 PROJECT:     PGA460 UART, TCI, & OWU Ultrasonic Time-of-Flight
 DESCRIPTION: Transmits and receives ultrasonic echo data to measure
              time-of-flight distance, width, and/or amplitude.
 CREATED:     22 February 2017
 UPDATED:     17 August 2017
 REVISION:    B
 AUTHOR:      A. Whitehead
 NOTES:       This example code is in the public domain.
*-------------------------------------------------------------------*/
#include <PGA460_USSC.h>

/*------------------------------------------------- run mode -----
|  userInputMode
|
|  Purpose:  This code can be operated in two run modes:
|    • userInputMode = allows the user to configure the device using
|      the COM serial terminal. Resulting data is printed in the
|      terminal view. Recommended run mode.
|    • standAloneMode = waits for the user to press the
|      LaucnhPad's PUSH2 button to automatically execute the
|      initializaiton routine, and begin the burst-and-listen captures.
|      The device is configured based on the hard-coded global
|      variables. LEDs are illumanted to represent approximate
|      object distance. Results also printed on serial COM terminal.
|      Comment out the run mode to use standAloneMode.
*-------------------------------------------------------------------*/
#define userInputMode

/*------------------------------------------------- Global Variables -----
|  Global Variables
|
|  Purpose:  Variables shared throughout the GetDistance sketch for
|    both userInput and standAlone modes. Hard-code these values to
|    the desired conditions when automatically updating the device
|    in standAlone mode.
*-------------------------------------------------------------------*/

// Configuration variables
  byte commMode = 0;            // Communication mode: 0=UART, 1=TCI, 2=OneWireUART
  byte fixedThr = 1;            // set P1 and P2 thresholds to 0=%25, 1=50%, or 2=75% of max;
initial minDistLim (i.e. 20cm) ignored
  byte xdcr = 1;                // set PGA460 to recommended settings for 0=Murata MA58MF14-
7N, 1=Murata MA40H1S-R
  byte agrTVG = 2;              // set TVG's analog front end gain range to 0=32-64dB, 1=46-
78dB, 2=52-84dB, or 3=58-90dB
  byte fixedTVG = 1;            // set fixed TVG level at 0=%25, 1=50%, or 1=75% of max
  byte runDiag = 0;             // run system diagnostics and temp/noise level before looping
burst+listen command
  byte edd = 0;                 // echo data dump of preset 1, 2, or neither
  byte burn = 0;                // trigger EE_CNTRL to burn and program user EEPROM memory
  byte cdMultiplier = 1;        // multiplier for command cycle delay
  byte numOfObj = 1;            // number of object to detect set to 1-8
  byte uartAddrUpdate = 0;      // PGA460 UART address to interface to; default is 0, possible
address 0-7
  bool objectDetected = false;  // object detected flag to break burst+listen cycle when true
  bool demoMode = false;        // only true when running UART/OWU multi device demo mode
  bool alwaysLong = false;      // always run preset 2, regardless of preset 1 result (hard-
coded only)
  double minDistLim = 0.1;      // minimum distance as limited by ringing decay of single
transducer and threshold masking
  uint16_t commandDelay = 0;    // Delay between each P1 and Preset 2 command
  uint32_t baudRate = 9600;     // UART baud rate: 9600, 19200, 38400, 57600, 74800, 115200

//PUSH BUTTON used for standAlone mode
  const int buttonPin = PUSH2;  // the number of the pushbutton pin
  int buttonState = 0;          // variable for reading the pushbutton status

// Result variables
  double distance = 0;          // one-way object distance in meters
  double width = 0;             // object width in microseconds
  double peak = 0;              // object peak in 8-bit
  double diagnostics = 0;       // diagnostic selector
  byte echoDataDumpElement = 0; // echo data dump element 0 to 127
  String interruptString = "";  // a string to hold incoming data
  boolean stringComplete = false; // whether the string is complete

// PGA460_USSC library class
  pga460 ussc;

/*------------------------------------------------- setup -----
|  function Setup
|
|  Purpose: (see funciton initPGA460 for details)
*-------------------------------------------------------------------*/
void setup() {                // put your setup code here, to run once
  initPGA460();
  }

/*------------------------------------------------- initPGA460 -----
|  function initPGA460
|
|  Purpose: One-time setup of PGA460-Q1 EVM hardware and software
|      in the following steps:
|    1) Configure the master to operate in UART, TCI, or OWU
|      communication mode.
|    2) Confgiure the EVM for compatiblity based on the selected
|      communicaton mode.
|    3) Option to update user EEPROM and threhsold registers with
|      pre-defined values.
|    4) Option to burn the EEPROM settings (not required unless
|      values are to be preserved after power cycling device).
|    5) Option to report echo data dump and/or system diagnostics.
|
|  In userInput mode, the user is prompted to enter values through
|   the Serial COM terminal to configure the device.
|
|  In standAlone mode, the user must hard-code the configuration
|   variables in the globals section for the device to
|   auto-configure in the background.
*-------------------------------------------------------------------*/
void initPGA460() {

    #ifdef userInputMode
    int inByte = 0;         // incoming serial byte

    Serial.begin(baudRate); // initialize COM UART serial channel
    delay(1000);
    Serial.println("PGA460-Q1 EVM UART/TCI/OWU Energia Demo for Ultrasonic Time-of-Flight");
    Serial.println("-------------------------------------------------------------------");
    Serial.println("Instructions: Configure the EVM by entering a byte value between 0-
9 or 'x' per request.");
    Serial.println("--- Input can be entered as a single string to auto-
increment/fill each request. E.g. 0011211000510");
    Serial.println("--- To skip the COM setup at any point, and use the hard-
coded values from thereon, enter a value of 's'.");
    Serial.println("--- To reset the program, and re-
prompt for user input, enter a value of 'q'.");

    int numInputs = 13;
    for (int i=0; i<numInputs; i++)
    {
      switch(i)
      {
        case 0: Serial.print("1. Communication Mode: 0=UART, 1=TCI, 2=OneWireUART ... "); break;

        case 1: Serial.print("2. UART kBaud: 0=9.6, 1=19.2, 2=38.4, 3=57.6, 4=74.8, 5=115.2 ...
"); break;
        case 2: Serial.print("3. P1 and P2 Thresholds: 0=%25, 1=50%, or 2=75% of max ... ");
break;
        case 3: Serial.print("4. Transducer Settings: 0=Murata MA58MF14-7N, 1=Murata MA40H1S-
R, x=Skip ... "); break;
        case 4: Serial.print("5. TVG Range: 0=32-64dB, 1=46-78dB, 2=52-84dB, or 3=58-
90dB, x=Skip ... "); break;
        case 5: Serial.print("6. Fixed TVG Level: 0=%25, 1=50%, or 2=75% of max, x=Skip ... ");
break;
        case 6: Serial.print("7. Minimum Distance = 0.1m * BYTE ... "); break;
        case 7: Serial.print("8. Run System Diagnostics?: 0=No, 1=Yes ... "); break;
        case 8: Serial.print("9. Echo Data Dump: 0=None, 1=P1BL, 2=P2BL, 3=P1LO, 4=P2LO,... ");
break;
        case 9: Serial.print("10. Burn User EEPROM?: 0=No, 1=Yes ... "); break;
        case 10: Serial.print("11. Command Cycle Delay: 10ms * BYTE ... "); break;
        case 11: Serial.print("12. Number of Objects to Detect (1-8) = BYTE ... "); break;
        case 12: Serial.print("13. UART Address of PGA460 (0-7) = BYTE ... "); break;
      }

    // only accept input as valid if 0-9, q, s, or x; otherwise, wait until valid input
      bool validInput = false;
      while (validInput == false)
      {
        while (Serial.available() == 0){}
        inByte = Serial.read();
         if (inByte==48 || inByte==49 || inByte==50 || inByte==51 ||
        inByte==52 || inByte==53 || inByte==54 || inByte==55 ||
        inByte==56 || inByte==57 || inByte==113 || inByte==115 || inByte==120)
        {
          validInput = true; // valid input, break while loop
        }
        else
        {
          // not a valid value
        }
      }

      //subtract 48d since ASCII '0' is 48d as a printable character
      inByte = inByte - 48;

      if (inByte != 115-48 && inByte != 113-48) // if input is neither 's' or 'q'
      {
        delay(300);
        Serial.println(inByte);
        switch(i)
        {
          case 0: commMode = inByte; break;
          case 1:
            switch(inByte)
              {
                case 0: baudRate=9600; break;
                case 1: baudRate=19200; break;
                case 2: baudRate=38400; break;
                case 3: baudRate=57600; break;
                case 4: baudRate=74800; break;
                case 5: baudRate=115200; break;
                default: baudRate=9600; break;
              }
          case 2: fixedThr = inByte; break;
          case 3: xdcr = inByte; break;
          case 4: agrTVG = inByte; break;
          case 5: fixedTVG = inByte; break;
          case 6: minDistLim = inByte * 0.1; break;
          case 7: runDiag = inByte; break;
          case 8: edd = inByte; break;
          case 9: burn = inByte; break;
          case 10: cdMultiplier = inByte; break;
          case 11: numOfObj = inByte; break;
          case 12: uartAddrUpdate = inByte; break;
          default: break;
        }
      }
      else if(inByte == 113-48) //  'q'
      {
        initPGA460(); // restart initializaiton routine
      }
      else //   's'
      {
        i=numInputs-1; // force for-loop to break
        Serial.println("");
      }
    }

    Serial.println("Configuring the PGA460 with the selected settings. Wait...");
    delay(300);

  #else  // standAlone mode
    pinMode(buttonPin, INPUT_PULLUP);         // initialize the pushbutton pin as an input
    while (digitalRead(buttonPin) == HIGH){}  // wait until user presses PUSH2 button to run
standalone mode
  #endif

/*------------------------------------------------- userInput & standAlone mode initialization ---
--
  Configure the EVM in the following order:
  1) Select PGA460 interface, device baud, and COM terminal baud up to 115.2k for targeted
address.
  2) Bulk write all threshold values to clear the THR_CRC_ERR.
  3) Bulk write user EEPROM with pre-define values in PGA460_USSC.c.
  4) Update analog front end gain range, and bulk write TVG.
  5) Run system diagnostics for frequency, decay, temperature, and noise measurements
  6) Program (burn) EEPROM memory to save user EEPROM values
  7) Run a preset 1 or 2 burst and/or listen command to capture the echo data dump

  if the input is 'x' (72d), then skip that configuration
*-------------------------------------------------------------------*/
  // -+-+-+-+-+-+-+-+-+-+- 1 : interface setup   -+-+-+-+-+-+-+-+-+-+- //
    ussc.initBoostXLPGA460(commMode, baudRate, uartAddrUpdate);
  // -+-+-+-+-+-+-+-+-+-+- 2 : bulk threshold write   -+-+-+-+-+-+-+-+-+-+- //
    if (fixedThr != 72){ussc.initThresholds(fixedThr);}
  // -+-+-+-+-+-+-+-+-+-+- 3 : bulk user EEPROM write   -+-+-+-+-+-+-+-+-+-+- //
    if (xdcr != 72){ussc.defaultPGA460(xdcr);}
  // -+-+-+-+-+-+-+-+-+-+- 4 : bulk TVG write   -+-+-+-+-+-+-+-+-+-+- //
    if (agrTVG != 72 && fixedTVG != 72){ussc.initTVG(agrTVG,fixedTVG);}
  // -+-+-+-+-+-+-+-+-+-+- 5 : run system diagnostics   -+-+-+-+-+-+-+-+-+-+- //
    if (runDiag == 1)
    {
      diagnostics = ussc.runDiagnostics(1,0);       // run and capture system diagnostics, and
print freq diag result
      Serial.print("System Diagnostics - Frequency (kHz): "); Serial.println(diagnostics);
      diagnostics = ussc.runDiagnostics(0,1);       // do not re-
run system diagnostic, but print decay diag result
      Serial.print("System Diagnostics - Decay Period (us): "); Serial.println(diagnostics);
      diagnostics = ussc.runDiagnostics(0,2);       // do not re-
run system diagnostic, but print temperature measurement
      Serial.print("System Diagnostics - Die Temperature (C): "); Serial.println(diagnostics);
      diagnostics = ussc.runDiagnostics(0,3);       // do not re-
run system diagnostic, but print noise level measurement
      Serial.print("System Diagnostics - Noise Level: "); Serial.println(diagnostics);
    }
  // -+-+-+-+-+-+-+-+-+-+- 6 : burn EEPROM   -+-+-+-+-+-+-+-+-+-+- //
    if(burn == 1)
    {
      byte burnStat = ussc.burnEEPROM();
      if(burnStat == true){Serial.println("EEPROM programmed successfully.");}
      else{Serial.println("EEPROM program failed.");}
    }
  // -+-+-+-+-+-+-+-+-+-+- 7 : capture echo data dump   -+-+-+-+-+-+-+-+-+-+- //
    if (edd != 0)                                   // run or skip echo data dump
    {
      Serial.println("Retrieving echo data dump profile. Wait...");
      ussc.runEchoDataDump(edd-
1);                  // run preset 1 or 2 burst and/or listen command
      for(int n=0; n<128; n++)                      // get all echo data dump results
        {
           echoDataDumpElement = ussc.pullEchoDataDump(n);
           Serial.print(echoDataDumpElement);
           Serial.print(",");
        }
      Serial.println("");
    }
  // -+-+-+-+-+-+-+-+-+-+-  others   -+-+-+-+-+-+-+-+-+-+- //
  commandDelay = 10 * cdMultiplier;                   // command cycle delay result in ms
  if (numOfObj == 0 || numOfObj >8) { numOfObj = 1; } // sets number of objects to detect to 1 if
invalid input

}

/*------------------------------------------------- main loop -----
|  main loop  GetDistance
|
|   The PGA460 is initiated with a Preset 1 Burst-and-Listen
|     Time-of-Flight measurement. Preset 1 is ideally configured for
|     short-range measurements (sub-1m range) when using the pre-defined
|     user EEPROM configurations.
|
|   If no object is detected, the PGA460 will then be issued a
|     Preset 2 Burst-and-Listen Time-of-Flight measurement.
|     Preset 2 is configured for long-range measurements (beyond
|     1m range).
|
|   Depending on the resulting distance, the diagnostics LEDs will
|     illuminate to represent a short, mid, or long range value.
|
|   In userInput mode, the distance, width, and/or amplitude value
|     of each object is serial printed on the COM terminal.
|
|   In standAlone mode, only distance can be represented visually
|     on the LEDs. The resulting values are still serial printed
|     on a COM terminal for debug, and to view the numerical values
|     of the data captured.
|
*-------------------------------------------------------------------*/

void loop() {                 // put your main code here, to run repeatedly
  while(1){
    // -+-+-+-+-+-+-+-+-+-+-  PRESET 1 (SHORT RANGE) MEASUREMENT   -+-+-+-+-+-+-+-+-+-+- //
      objectDetected = false;                       // Initialize object detected flag to false
      ussc.ultrasonicCmd(0,numOfObj);               // run preset 1 (short distance) burst+listen
for 1 object
      ussc.pullUltrasonicMeasResult(demoMode);      // Pull Ultrasonic Measurement Result
      for (byte i=0; i<numOfObj; i++)
      {
        // Log uUltrasonic Measurement Result: Obj1: 0=Distance(m), 1=Width, 2=Amplitude; Obj2:
3=Distance(m), 4=Width, 5=Amplitude; etc.;
          distance = ussc.printUltrasonicMeasResult(0+(i*3));
          //width = ussc.printUltrasonicMeasResult(1+(i*3));
          //peak = ussc.printUltrasonicMeasResult(2+(i*3));

        delay(commandDelay);

        if (distance > minDistLim && distance < 11.2)  // turn on DS1_LED if object is above
minDistLim
        {
            ussc.toggleLEDs(HIGH,LOW,LOW);
            Serial.print("P1 Obj"); Serial.print(i+1); Serial.print(" Distance (m): ");
Serial.println(distance);
            objectDetected = true;
        }
      }

    // -+-+-+-+-+-+-+-+-+-+-  PRESET 2 (LONG RANGE) MEASUREMENT   -+-+-+-+-+-+-+-+-+-+- //
      if(objectDetected == false || alwaysLong == true)                       // If no preset 1
(short distance) measurement result, switch to Preset 2 B+L command
      {
        ussc.ultrasonicCmd(1,numOfObj);                // run preset 2 (long distance)
burst+listen for 1 object
        ussc.pullUltrasonicMeasResult(demoMode);                // Get Ultrasonic Measurement
Result
        for (byte i=0; i<numOfObj; i++)
        {
          distance = ussc.printUltrasonicMeasResult(0+(i*3));   // Print Ultrasonic Measurement
Result i.e. Obj1: 0=Distance(m), 1=Width, 2=Amplitude; Obj2: 3=Distance(m), 4=Width, 5=Amplitude;
          //width = ussc.printUltrasonicMeasResult(1+(i*3));
          //peak = ussc.printUltrasonicMeasResult(2+(i*3));

          delay(commandDelay);

          if (distance < 1 && distance > minDistLim)    // turn on DS1_LED and F_DIAG_LED if
object is within 1m
          {
              ussc.toggleLEDs(HIGH,LOW,LOW);
              Serial.print("P2 Obj"); Serial.print(i+1); Serial.print(" Distance (m): ");
Serial.println(distance);
              objectDetected = true;
          }
          else if (distance < 3 && distance >= 1)      // turn on DS1_LED and F_DIAG_LED if
object is within 3m
          {
              ussc.toggleLEDs(HIGH,HIGH,LOW);
              Serial.print("P2 Obj"); Serial.print(i+1); Serial.print(" Distance (m): ");
Serial.println(distance);
              objectDetected = true;
          }
          else if (distance >= 3 && distance < 11.2)     // turn on DS1_LED, F_DIAG_LED, and 
V_DIAG_LED if object is greater than 3m
          {
              ussc.toggleLEDs(HIGH,HIGH,HIGH);
              Serial.print("P2 Obj"); Serial.print(i+1); Serial.print(" Distance (m): "); 
Serial.println(distance);
              objectDetected = true;
          }
          else if (distance == 0)                         // turn off all LEDs if no object 
detected
          {
              ussc.toggleLEDs(LOW,LOW,LOW);
              //Serial.print("Error reading measurement results..."); //Serial.println(distance);
          }
          else //(distance > 11.2 && distance < minDistLim)         // turn off all LEDs if no 
object detected or below minimum distance limit
          {
              if (i == numOfObj-1 && objectDetected == false)
              {
                ussc.toggleLEDs(LOW,LOW,LOW);
                Serial.println("No object...");
              }
          }
        }
      }

    // -+-+-+-+-+-+-+-+-+-+-  STATUS   -+-+-+-+-+-+-+-+-+-+- //
      digitalWrite(GREEN_LED, !digitalRead(GREEN_LED));   //toggle green LED after each sequence
      digitalWrite(RED_LED, !digitalRead(GREEN_LED));     //toggle red LED after each sequence

    // -+-+-+-+-+-+-+-+-+-+-  SERIAL MONITORING   -+-+-+-+-+-+-+-+-+-+- //
      // Check for serial character at COM terminal
      while (Serial.available())
      {
       char inChar = (char)Serial.read(); // get the new byte
       // if the incoming character is a 'q', set a flag, stop the main loop, and re-
run initialization
        if (inChar == 'q'){stringComplete = true; initPGA460();}
     }
  }
}
```

5 PGA460 Energia Library

5.1 *pga460_ussc.cpp*

The *pga460_ussc.cpp* code is as follows:

```
/*
  PGA460_USSC.cpp
  Created by A. Whitehead <make@energia.nu>, Initial: Nov 2016, Updated: Aug 2017

  //Released into the public domain.
*/
#include "PGA460_USSC.h"
#include "Energia.h"

/*------------------------------------------------- Global Variables -----
 |  Global Variables
 |
 |  Purpose:  Variables shared throughout the PGA460_USSC.cpp functions
 *-------------------------------------------------------------------*/
#pragma region globals
// Pin mapping of BOOSTXL-PGA460 to LaunchPad by pin name
    #define DECPL_A 2
    #define RXD_LP 3
    #define TXD_LP 4
    #define DECPL_D 5
    #define TEST_A 6
    #define TCI_CLK 7
    #define TEST_D 8
    #define MEM_SOMI 9
    #define MEM_SIMO 10
    #define TCI_RX 14
    #define TCI_TX 15
    #define COM_SEL 17
    #define COM_PD 18
    #define SCLK_CLK 34
    #define MEM_HOLD 36
    #define MEM_CS 37
    #define DS1_LED 38
    #define F_DIAG_LED 39
    #define V_DIAG_LED 40

// Serial read timeout in milliseconds
    #define MAX_MILLIS_TO_WAIT 250

// Define UART commands by name
    // Single Address
        byte P1BL = 0x00;
        byte P2BL = 0x01;
        byte P1LO = 0x02;
        byte P2LO = 0x03;
        byte TNLM = 0x04;
        byte UMR = 0x05;
        byte TNLR = 0x06;
        byte TEDD = 0x07;
        byte SD = 0x08;
        byte SRR = 0x09;
        byte SRW = 0x0A;
        byte EEBR = 0x0B;
        byte EEBW = 0x0C;
        byte TVGBR = 0x0D;
        byte TVGBW = 0x0E;
        byte THRBR = 0x0F;
        byte THRBW = 0x10;
    //Broadcast
        byte BC_P1BL = 0x11;
```

```
            byte BC_P2BL = 0x12;
            byte BC_P1LO = 0x13;
            byte BC_P2LO = 0x14;
            byte BC_TNLM = 0x15;
            byte BC_RW = 0x16;
            byte BC_EEBW = 0x17;
            byte BC_TVGBW = 0x18;
            byte BC_THRBW = 0x19;
            //CMDs 26-31 are reserved

    // List user registers by name with default settings from TI factory
        byte USER_DATA1 = 0x00;
        byte USER_DATA2 = 0x00;
        byte USER_DATA3 = 0x00;
        byte USER_DATA4 = 0x00;
        byte USER_DATA5 = 0x00;
        byte USER_DATA6 = 0x00;
        byte USER_DATA7 = 0x00;
        byte USER_DATA8 = 0x00;
        byte USER_DATA9 = 0x00;
        byte USER_DATA10 = 0x00;
        byte USER_DATA11 = 0x00;
        byte USER_DATA12 = 0x00;
        byte USER_DATA13 = 0x00;
        byte USER_DATA14 = 0x00;
        byte USER_DATA15 = 0x00;
        byte USER_DATA16 = 0x00;
        byte USER_DATA17 = 0x00;
        byte USER_DATA18 = 0x00;
        byte USER_DATA19 = 0x00;
        byte USER_DATA20 = 0x00;
        byte TVGAIN0 = 0xAF;
        byte TVGAIN1 = 0xFF;
        byte TVGAIN2 = 0xFF;
        byte TVGAIN3 = 0x2D;
        byte TVGAIN4 = 0x68;
        byte TVGAIN5 = 0x36;
        byte TVGAIN6 = 0xFC;
        byte INIT_GAIN = 0xC0;
        byte FREQUENCY  = 0x8C;
        byte DEADTIME = 0x00;
        byte PULSE_P1 = 0x01;
        byte PULSE_P2 = 0x12;
        byte CURR_LIM_P1 = 0x47;
        byte CURR_LIM_P2 = 0xFF;
        byte REC_LENGTH = 0x1C;
        byte FREQ_DIAG = 0x00;
        byte SAT_FDIAG_TH = 0xEE;
        byte FVOLT_DEC = 0x7C;
        byte DECPL_TEMP = 0x0A;
        byte DSP_SCALE = 0x00;
        byte TEMP_TRIM = 0x00;
        byte P1_GAIN_CTRL = 0x00;
        byte P2_GAIN_CTRL = 0x00;
        byte EE_CRC = 0xFF;
        byte EE_CNTRL = 0x00;
        byte P1_THR_0 = 0x88;
        byte P1_THR_1 = 0x88;
        byte P1_THR_2 = 0x88;
        byte P1_THR_3 = 0x88;
        byte P1_THR_4 = 0x88;
        byte P1_THR_5 = 0x88;
        byte P1_THR_6 = 0x84;
        byte P1_THR_7 = 0x21;
        byte P1_THR_8 = 0x08;
        byte P1_THR_9 = 0x42;
        byte P1_THR_10 = 0x10;
        byte P1_THR_11 = 0x80;
        byte P1_THR_12 = 0x80;
        byte P1_THR_13 = 0x80;
        byte P1_THR_14 = 0x80;
        byte P1_THR_15 = 0x80;
        byte P2_THR_0 = 0x88;
        byte P2_THR_1 = 0x88;
        byte P2_THR_2 = 0x88;
        byte P2_THR_3 = 0x88;
        byte P2_THR_4 = 0x88;
        byte P2_THR_5 = 0x88;
        byte P2_THR_6 = 0x84;
        byte P2_THR_7 = 0x21;
        byte P2_THR_8 = 0x08;
        byte P2_THR_9 = 0x42;
        byte P2_THR_10 = 0x10;
        byte P2_THR_11 = 0x80;
        byte P2_THR_12 = 0x80;
        byte P2_THR_13 = 0x80;
        byte P2_THR_14 = 0x80;
        byte P2_THR_15 = 0x80;

    // Miscellaneous variables; (+) indicates OWU transmitted byte offset
        byte checksum = 0x00;             // UART checksum value
        byte ChecksumInput[44];         // data byte array for checksum calculator
        byte ultraMeasResult[34+3];     // data byte array for cmd5 and tciB+L return
        byte diagMeasResult[5+3];         // data byte array for cmd8 and index1 return
        byte tempNoiseMeasResult[4+3];     // data byte array for cmd6 and index0&1 return
        byte echoDataDump[130+3];         // data byte array for cmd7 and index12 return
        byte tempOrNoise = 0;             // data byte to determine if temp or noise measurement is
    to be performed
        byte comm = 0;                     // indicates UART (0), TCI (1), OWU (2) communication
    mode
        unsigned long starttime;         // used for function time out
        //UART & OWU exclusive variables
            byte syncByte = 0x55;         // data byte for Sync field set UART baud rate of PGA460
            byte regAddr = 0x00;         // data byte for Register Address
            byte regData = 0x00;         // data byte for Register Data
            byte uartAddr = 0;             // PGA460 UART device address (0-
    7). '0' is factory default address
            byte numObj = 1;             // number of objects to detect
            //OWU exclusive variables
                signed int owuShift = 0;// accoutns for OWU receiver buffer offset for capturing
    master transmitted data - always 0 for standard two-wire UART
        //TCI exclusive variables
            byte bufRecv[128];                 // TCI receive data buffer for all commands
            unsigned long tciToggle;        // used to log TCI burst+listen time of object
            unsigned int objTime[8];        // array to capture up to eight object TCI burst+listen
    toggles
    #pragma endregion globals

    /*------------------------------------------------- PGA460 Top Level -----
     |  PGA460 Top Level Scope Resolution Operator
     |
     | Use the double colon operator (::) to qualify a C++ member function, a top
     | level function, or a variable with global scope with:
     | • An overloaded name (same name used with different argument types)
     | • An ambiguous name (same name used in different classes)
     *-------------------------------------------------------------------*/
    pga460::pga460(){}

    /*------------------------------------------------- initBoostXLPGA460 -----
     |  Function initBoostXLPGA460
     |
     |  Purpose:  Configure the master communication mode and BOOSTXL-
```

```
PGA460 hardware to operate in UART, TCI, or OWU mode.
 |  Configures master serial baud rate for UART/OWU modes. Updates UART address based on sketch
input.
 |
 |  Parameters:
 |        mode (IN) -- sets communicaiton mode.
 |            0=UART
 |            1=TCI
 |            2=OWU
 |            6=Bus_Demo_Bulk_TVG_or_Threshold_Broadcast_is_True
 |            7=Bus_Demo_UART_Mode
 |            8=Bus_Demo_OWU_One_Time_Setup
 |            9=Bus_Demo_OWU_Mode
 |         baud (IN) -- PGA460 accepts a baud rate of 9600 to 115.2k bps
 |         uartAddrUpdate (IN) -- PGA460 address range from 0 to 7
 |
 |  Returns:  none
 *--------------------------------------------------------------------*/
void pga460::initBoostXLPGA460(byte mode, uint32_t baud, byte uartAddrUpdate)
{
    // check for valid UART address
    if (uartAddrUpdate > 7)
    {
        uartAddrUpdate = 0; // default to '0'
        Serial.println("ERROR - Invalid UART Address!");
    }
    // globally update target PGA460 UART address and commands
    if (uartAddr != uartAddrUpdate)
    {
        // Update commands to account for new UART addr
          // Single Address
           P1BL = 0x00 + (uartAddrUpdate << 5);
           P2BL = 0x01 + (uartAddrUpdate << 5);
           P1LO = 0x02 + (uartAddrUpdate << 5);
           P2LO = 0x03 + (uartAddrUpdate << 5);
           TNLM = 0x04 + (uartAddrUpdate << 5);
           UMR = 0x05 + (uartAddrUpdate << 5);
           TNLR = 0x06 + (uartAddrUpdate << 5);
           TEDD = 0x07 + (uartAddrUpdate << 5);
           SD = 0x08 + (uartAddrUpdate << 5);
           SRR = 0x09 + (uartAddrUpdate << 5);
           SRW = 0x0A + (uartAddrUpdate << 5);
           EEBR = 0x0B + (uartAddrUpdate << 5);
           EEBW = 0x0C + (uartAddrUpdate << 5);
           TVGBR = 0x0D + (uartAddrUpdate << 5);
           TVGBW = 0x0E + (uartAddrUpdate << 5);
           THRBR = 0x0F + (uartAddrUpdate << 5);
           THRBW = 0x10 + (uartAddrUpdate << 5);
    }
    uartAddr = uartAddrUpdate;

    // turn on LP's Red LED to indicate code has started to run
    pinMode(RED_LED, OUTPUT); digitalWrite(RED_LED, HIGH);

    // turn off BOOSTXL-PGA460's diagnostic LEDs
    pinMode(DS1_LED, OUTPUT); digitalWrite(DS1_LED, LOW);
    pinMode(F_DIAG_LED, OUTPUT); digitalWrite(F_DIAG_LED, LOW);
    pinMode(V_DIAG_LED, OUTPUT); digitalWrite(V_DIAG_LED, LOW);

    // set communication mode flag
    if (mode < 3)
    {
        comm = mode;
        // disable synchronous mode dump to external memory
        pinMode(MEM_HOLD, OUTPUT);  digitalWrite(MEM_HOLD, HIGH);
        pinMode(MEM_CS, OUTPUT);  digitalWrite(MEM_CS, HIGH);
    }
    else if (mode == 6)
    {
        comm = 6; // bus demo user input mode only, and threshold or TVG bulk write broadcast
commands are true
    }
    else if ((mode == 7) || (mode == 9))
    {
        comm = mode - 7; // bus demo only for either UART or OWU mode
    }
    else
    {
        comm = 99; // invalid communication type
    }

    switch (mode)
    {
        case 0: // UART Mode
            // enable PGA460 UART communication mode
            pinMode(COM_PD, OUTPUT);  digitalWrite(COM_PD, LOW);
            pinMode(COM_SEL, OUTPUT);  digitalWrite(COM_SEL, LOW);
            Serial.begin(baud);     // initialize COM UART serial channel
            Serial1.begin(baud);    // initialize PGA460 UART serial channel
            break;
        case 1: //TCI Mode
            // enable PGA460 TCI communication mode
            pinMode(COM_PD, OUTPUT);  digitalWrite(COM_PD, LOW);
            pinMode(COM_SEL, OUTPUT);  digitalWrite(COM_SEL, LOW);
            pinMode(TCI_TX, OUTPUT); digitalWrite(TCI_TX, HIGH);
            pinMode(TCI_RX, INPUT_PULLUP);
            Serial.begin(baud);     // initialize COM UART serial channel
            Serial1.begin(baud);    // initialize PGA460 UART serial channel //DEBUG remove
            break;
        case 2: //OWU setup (part I)
            // enable PGA460 UART communication mode
            pinMode(COM_PD, OUTPUT);  digitalWrite(COM_PD, LOW);
            pinMode(COM_SEL, OUTPUT);  digitalWrite(COM_SEL, LOW);
            Serial.begin(baud);     // initialize COM UART serial channel
            Serial1.begin(baud);    // initialize PGA460 UART serial channel
            PULSE_P1 = 0x80 | PULSE_P1; // update IO_IF_SEL bit to '1' for OWU mode for bulk
EEPROM write
            break;
        default: break;
    }

    //OWU setup (part II)
    if ((comm == 2) || (mode == 8)) // mode8 is for one time setup of OWU per slave device for
bus demo
    {
        // UART write to register PULSE_P1 (addr 0x1E) to set device into OWU mode
        regAddr = 0x1E;
        regData = PULSE_P1;
        byte buf10[5] = {syncByte, SRW, regAddr, regData, calcChecksum(SRW)};
        Serial1.write(buf10, sizeof(buf10));
        delay(50);

        // enable PGA460 OWU communication mode
        pinMode(COM_SEL, OUTPUT);  digitalWrite(COM_SEL, HIGH);
    }

    pinMode(GREEN_LED, OUTPUT);
    if ((mode == 7) || (mode == 9))
    {
        // do not delay bus demo loop by blinking Green LED
        digitalWrite(RED_LED, LOW); // turn off LaunchPad's Red LED
    }
    else // blink LP's Green LED twice to indicate initialization is complete
    {
        digitalWrite(GREEN_LED, HIGH);
        for(int loops = 0; loops < 5; loops++)
        {
            digitalWrite(GREEN_LED, HIGH);
            delay(200);
            digitalWrite(GREEN_LED, LOW);
            delay(200);
        }
    }

    return;
}

/*------------------------------------------------- defaultPGA460 -----
 |  Function defaultPGA460
 |
 |  Purpose:  Updates user EEPROM values, and performs bulk EEPROM write.
 |
 |  Parameters:
 |        xdcr (IN) -- updates user EEPROM based on predefined listing for a specific transducer.
 |            Modify existing case statements, or append additional case-
statement for custom user EEPROM configurations.
 |            • 0 = Murata MA58MF14-7N
 |            • 1 = Murata MA40H1S-R
 |
 |  Returns:  none
 *-------------------------------------------------------------------*/
void pga460::defaultPGA460(byte xdcr)
{
    switch (xdcr)
    {
        case 0: // Murata MA58MF14-7N
           USER_DATA1 = 0x00;
           USER_DATA2 = 0x00;
           USER_DATA3 = 0x00;
           USER_DATA4 = 0x00;
           USER_DATA5 = 0x00;
           USER_DATA6 = 0x00;
           USER_DATA7 = 0x00;
           USER_DATA8 = 0x00;
           USER_DATA9 = 0x00;
           USER_DATA10 = 0x00;
           USER_DATA11 = 0x00;
           USER_DATA12 = 0x00;
           USER_DATA13 = 0x00;
           USER_DATA14 = 0x00;
           USER_DATA15 = 0x00;
           USER_DATA16 = 0x00;
           USER_DATA17 = 0x00;
           USER_DATA18 = 0x00;
           USER_DATA19 = 0x00;
           USER_DATA20 = 0x00;
           TVGAIN0 = 0xAA;
           TVGAIN1 = 0xAA;
           TVGAIN2 = 0xAA;
           TVGAIN3 = 0x82;
           TVGAIN4 = 0x08;
           TVGAIN5 = 0x20;
           TVGAIN6 = 0x80;
           INIT_GAIN = 0x60;
           FREQUENCY  = 0x8F;
           DEADTIME = 0xA0;
           if (comm == 2)
           {
                PULSE_P1 = 0x80 | 0x04;
           }
           else
           {
                PULSE_P1 = 0x04;
           }
           PULSE_P2 = 0x10;
           CURR_LIM_P1 = 0x55;
           CURR_LIM_P2 = 0x55;
           REC_LENGTH = 0x19;
           FREQ_DIAG = 0x33;
           SAT_FDIAG_TH = 0xEE;
           FVOLT_DEC = 0x7C;
           DECPL_TEMP = 0x4F;
           DSP_SCALE = 0x00;
           TEMP_TRIM = 0x00;
           P1_GAIN_CTRL = 0x09;
           P2_GAIN_CTRL = 0x09;
            break;
        case 1: // Murata MA40H1SR
           USER_DATA1 = 0x00;
           USER_DATA2 = 0x00;
           USER_DATA3 = 0x00;
           USER_DATA4 = 0x00;
           USER_DATA5 = 0x00;
           USER_DATA6 = 0x00;
           USER_DATA7 = 0x00;
           USER_DATA8 = 0x00;
           USER_DATA9 = 0x00;
           USER_DATA10 = 0x00;
           USER_DATA11 = 0x00;
           USER_DATA12 = 0x00;
           USER_DATA13 = 0x00;
           USER_DATA14 = 0x00;
           USER_DATA15 = 0x00;
           USER_DATA16 = 0x00;
           USER_DATA17 = 0x00;
           USER_DATA18 = 0x00;
           USER_DATA19 = 0x00;
           USER_DATA20 = 0x00;
           TVGAIN0 = 0xAA;
           TVGAIN1 = 0xAA;
           TVGAIN2 = 0xAA;
           TVGAIN3 = 0x51;
           TVGAIN4 = 0x45;
           TVGAIN5 = 0x14;
           TVGAIN6 = 0x50;
           INIT_GAIN = 0x54;
           FREQUENCY  = 0x32;
           DEADTIME = 0xA0;
           if (comm == 2)
           {
                PULSE_P1 = 0x80 | 0x08;
           }
           else
           {
                PULSE_P1 = 0x08;
           }
           PULSE_P2 = 0x10;
           CURR_LIM_P1 = 0x40;
           CURR_LIM_P2 = 0x40;
           REC_LENGTH = 0x19;
           FREQ_DIAG = 0x33;
           SAT_FDIAG_TH = 0xEE;
           FVOLT_DEC = 0x7C;
           DECPL_TEMP = 0x4F;
           DSP_SCALE = 0x00;
           TEMP_TRIM = 0x00;
           P1_GAIN_CTRL = 0x09;
           P2_GAIN_CTRL = 0x09;
            break;
        case 2: // user custom
        {
            // insert custom user EEPROM listing
        }
        default: break;
    }

        if ((comm !=1) && (comm !=6)) // UART or OWU mode and not busDemo6
        {
            byte buf12[46] = {syncByte, EEBW, USER_DATA1, USER_DATA2, USER_DATA3, USER_DATA4,
USER_DATA5, USER_DATA6,
                USER_DATA7, USER_DATA8, USER_DATA9, USER_DATA10, USER_DATA11, USER_DATA12,
USER_DATA13, USER_DATA14,
                USER_DATA15,USER_DATA16,USER_DATA17,USER_DATA18,USER_DATA19,USER_DATA20,

TVGAIN0,TVGAIN1,TVGAIN2,TVGAIN3,TVGAIN4,TVGAIN5,TVGAIN6,INIT_GAIN,FREQUENCY,DEADTIME,

PULSE_P1,PULSE_P2,CURR_LIM_P1,CURR_LIM_P2,REC_LENGTH,FREQ_DIAG,SAT_FDIAG_TH,FVOLT_DEC,DECPL_TEMP,
                DSP_SCALE,TEMP_TRIM,P1_GAIN_CTRL,P2_GAIN_CTRL,calcChecksum(EEBW)};

            Serial1.write(buf12, sizeof(buf12)); // serial transmit master data for bulk EEPROM
            delay(50);

            // Update targeted UART_ADDR to address defined in EEPROM bulk switch-case
            byte uartAddrUpdate = (PULSE_P2 >> 5) & 0x07;
            if (uartAddr != uartAddrUpdate)
            {
                // Update commands to account for new UART addr
                  // Single Address
                   P1BL = 0x00 + (uartAddrUpdate << 5);
                   P2BL = 0x01 + (uartAddrUpdate << 5);
                   P1LO = 0x02 + (uartAddrUpdate << 5);
                   P2LO = 0x03 + (uartAddrUpdate << 5);
                   TNLM = 0x04 + (uartAddrUpdate << 5);
                   UMR = 0x05 + (uartAddrUpdate << 5);
                   TNLR = 0x06 + (uartAddrUpdate << 5);
                   TEDD = 0x07 + (uartAddrUpdate << 5);
                   SD = 0x08 + (uartAddrUpdate << 5);
                   SRR = 0x09 + (uartAddrUpdate << 5);
                   SRW = 0x0A + (uartAddrUpdate << 5);
                   EEBR = 0x0B + (uartAddrUpdate << 5);
                   EEBW = 0x0C + (uartAddrUpdate << 5);
                   TVGBR = 0x0D + (uartAddrUpdate << 5);
                   TVGBW = 0x0E + (uartAddrUpdate << 5);
                   THRBR = 0x0F + (uartAddrUpdate << 5);
                   THRBW = 0x10 + (uartAddrUpdate << 5);
            }
            uartAddr = uartAddrUpdate;
        }
        else if (comm == 6)
        {
            return;
        }
        else
        {
            tciIndexRW(13, true);    // TCI index 13 write
        }

    return;
}
```

```
/*------------------------------------------------- initThresholds -----
 |  Function initThresholds
 |
 |  Purpose:  Updates threshold mapping for both presets, and performs bulk threshold write
 |
 |  Parameters:
 |        thr (IN) --
 updates all threshold levels to a fixed level based on specific percentage of the maximum level.
 |            All times are mid-code (1.4ms intervals).
 |            Modify existing case statements, or append additional case-
statement for custom user threshold configurations.
 |            • 0 = 25% Levels 64 of 255
 |            • 1 = 50% Levels 128 of 255
 |            • 2 = 75% Levels 192 of 255
 |
 |  Returns:  none
 *-------------------------------------------------------------------*/
void pga460::initThresholds(byte thr)
{
    switch (thr)
    {
        case 0: //25% Levels 64 of 255
           P1_THR_0 = 0x88;
           P1_THR_1 = 0x88;
           P1_THR_2 = 0x88;
           P1_THR_3 = 0x88;
           P1_THR_4 = 0x88;
           P1_THR_5 = 0x88;
           P1_THR_6 = 0x42;
           P1_THR_7 = 0x10;
           P1_THR_8 = 0x84;
           P1_THR_9 = 0x21;
           P1_THR_10 = 0x08;
           P1_THR_11 = 0x40;
           P1_THR_12 = 0x40;
           P1_THR_13 = 0x40;
           P1_THR_14 = 0x40;
           P1_THR_15 = 0x00;
           P2_THR_0 = 0x88;
           P2_THR_1 = 0x88;
           P2_THR_2 = 0x88;
           P2_THR_3 = 0x88;
           P2_THR_4 = 0x88;
           P2_THR_5 = 0x88;
           P2_THR_6 = 0x42;
           P2_THR_7 = 0x10;
           P2_THR_8 = 0x84;
           P2_THR_9 = 0x21;
           P2_THR_10 = 0x08;
           P2_THR_11 = 0x40;
           P2_THR_12 = 0x40;
           P2_THR_13 = 0x40;
           P2_THR_14 = 0x40;
           P2_THR_15 = 0x00;
        break;

        case 1: //50% Level (midcode) 128 of 255
           P1_THR_0 = 0x88;
           P1_THR_1 = 0x88;
           P1_THR_2 = 0x88;
           P1_THR_3 = 0x88;
           P1_THR_4 = 0x88;
           P1_THR_5 = 0x88;
           P1_THR_6 = 0x84;
           P1_THR_7 = 0x21;
           P1_THR_8 = 0x42;
           P1_THR_9 = 0x10;
           P1_THR_10 = 0x10;
           P1_THR_11 = 0x80;
           P1_THR_12 = 0x80;
           P1_THR_13 = 0x80;
           P1_THR_14 = 0x80;
           P1_THR_15 = 0x00;
           P2_THR_0 = 0x88;
           P2_THR_1 = 0x88;
           P2_THR_2 = 0x88;
           P2_THR_3 = 0x88;
           P2_THR_4 = 0x88;
           P2_THR_5 = 0x88;
           P2_THR_6 = 0x84;
           P2_THR_7 = 0x21;
           P2_THR_8 = 0x42;
           P2_THR_9 = 0x10;
           P2_THR_10 = 0x10;
           P2_THR_11 = 0x80;
           P2_THR_12 = 0x80;
           P2_THR_13 = 0x80;
           P2_THR_14 = 0x80;
           P2_THR_15 = 0x00;
        break;

        case 2: //75% Levels 192 of 255
           P1_THR_0 = 0x88;
           P1_THR_1 = 0x88;
           P1_THR_2 = 0x88;
           P1_THR_3 = 0x88;
           P1_THR_4 = 0x88;
           P1_THR_5 = 0x88;
           P1_THR_6 = 0xC6;
           P1_THR_7 = 0x31;
           P1_THR_8 = 0x8C;
           P1_THR_9 = 0x63;
           P1_THR_10 = 0x18;
           P1_THR_11 = 0xC0;
           P1_THR_12 = 0xC0;
           P1_THR_13 = 0xC0;
           P1_THR_14 = 0xC0;
           P1_THR_15 = 0x00;
           P2_THR_0 = 0x88;
           P2_THR_1 = 0x88;
           P2_THR_2 = 0x88;
           P2_THR_3 = 0x88;
           P2_THR_4 = 0x88;
           P2_THR_5 = 0x88;
           P2_THR_6 = 0xC6;
           P2_THR_7 = 0x31;
           P2_THR_8 = 0x8C;
           P2_THR_9 = 0x63;
           P2_THR_10 = 0x18;
           P2_THR_11 = 0xC0;
           P2_THR_12 = 0xC0;
           P2_THR_13 = 0xC0;
           P2_THR_14 = 0xC0;
           P2_THR_15 = 0x00;
        break;

        default: break;
    }

    if ((comm !=1) && (comm !=6))     // UART or OWU mode and not busDemo6
    {
        byte buf16[35] = {syncByte, THRBW, P1_THR_0, P1_THR_1, P1_THR_2, P1_THR_3, P1_THR_4,
P1_THR_5, P1_THR_6,
              P1_THR_7, P1_THR_8, P1_THR_9, P1_THR_10, P1_THR_11, P1_THR_12, P1_THR_13,
P1_THR_14, P1_THR_15,
              P2_THR_0, P2_THR_1, P2_THR_2, P2_THR_3, P2_THR_4, P2_THR_5, P2_THR_6,
              P2_THR_7, P2_THR_8, P2_THR_9, P2_THR_10, P2_THR_11, P2_THR_12, P2_THR_13,
P2_THR_14, P2_THR_15,
              calcChecksum(THRBW)};
        Serial1.write(buf16, sizeof(buf16)); // serial transmit master data for bulk threhsold
    }
    else if(comm == 6)
    {
        return;
    }
    else
    {
        tciIndexRW(5, true); //TCI Threshold Preset 1 write
        tciIndexRW(6, true); //TCI Threshold Preset 2 write
    }

    delay(100);
    return;
}

/*------------------------------------------------- initTVG -----
 |  Function initTVG
 |
 |  Purpose:  Updates time varying gain (TVG) range and mapping, and performs bulk TVG write
 |
 |  Parameters:
 |        agr (IN) -- updates the analog gain range for the TVG.
 |            • 0 = 32-64dB
 |            • 1 = 46-78dB
 |            • 2 = 52-84dB
 |            • 3 = 58-90dB
 |        tvg (IN) --
 updates all TVG levels to a fixed level based on specific percentage of the maximum level.
 |            All times are mid-code (2.4ms intervals).
 |            Modify existing case statements, or append additional case-
statement for custom user TVG configurations
 |            • 0 = 25% Levels of range
 |            • 1 = 50% Levels of range
 |            • 2 = 75% Levels of range
 |
 |  Returns:  none
 *-------------------------------------------------------------------*/
void pga460::initTVG(byte agr, byte tvg)
{
    byte gain_range = 0x4F;
    // set AFE gain range
    switch (agr)
    {
        case 3: //58-90dB
            gain_range =  0x0F;
            break;
        case 2: //52-84dB
            gain_range = 0x4F;
            break;
        case 1: //46-78dB
            gain_range = 0x8F;
            break;
        case 0: //32-64dB
            gain_range = 0xCF;
            break;
        default: break;
    }
```

```
          if ((comm !=1) && (comm !=6))     // UART or OWU mode and not busDemo6
          {
              regAddr = 0x26;
              regData = gain_range;
              byte buf10[5] = {syncByte, SRW, regAddr, regData, calcChecksum(SRW)};
              Serial1.write(buf10, sizeof(buf10));
          }
          else if(comm == 6)
          {
              return;
          }
          else
          {
              //TODO enable index 10 write
              //tciIndexRW(10, true);
          }


          //Set fixed AFE gain value
          switch (tvg)
          {
              case 0: //25% Level
                 TVGAIN0 = 0x88;
                 TVGAIN1 = 0x88;
                 TVGAIN2 = 0x88;
                 TVGAIN3 = 0x41;
                 TVGAIN4 = 0x04;
                 TVGAIN5 = 0x10;
                 TVGAIN6 = 0x40;
              break;

              case 1: //50% Levels
                 TVGAIN0 = 0x88;
                 TVGAIN1 = 0x88;
                 TVGAIN2 = 0x88;
                 TVGAIN3 = 0x82;
                 TVGAIN4 = 0x08;
                 TVGAIN5 = 0x20;
                 TVGAIN6 = 0x80;
              break;

              case 2: //75% Levels
                 TVGAIN0 = 0x88;
                 TVGAIN1 = 0x88;
                 TVGAIN2 = 0x88;
                 TVGAIN3 = 0xC3;
                 TVGAIN4 = 0x0C;
                 TVGAIN5 = 0x30;
                 TVGAIN6 = 0xC0;
              break;

              default: break;
          }

          if ((comm !=1) && (comm !=6))     // UART or OWU mode and not busDemo6
          {
              byte buf14[10] = {syncByte, TVGBW, TVGAIN0, TVGAIN1, TVGAIN2, TVGAIN3, TVGAIN4, TVGAIN5,
      TVGAIN6, calcChecksum(TVGBW)};
              Serial1.write(buf14, sizeof(buf14)); // serial transmit master data for bulk TVG
          }
          else if(comm == 6)
          {
              return;
          }
          else
          {
```

```
        tciIndexRW(8, true); //TCI bulk TVG write
    }

    return;
}

/*------------------------------------------------- ultrasonicCmd -----
 |  Function ultrasonicCmd
 |
 |  Purpose:  Issues a burst-and-listen or listen-
only command based on the number of objects to be detected.
 |
 |  Parameters:
 |        cmd (IN) -- determines which preset command is run
 |              • 0 = Preset 1 Burst + Listen command
 |              • 1 = Preset 2 Burst + Listen command
 |              • 2 = Preset 1 Listen Only command
 |              • 3 = Preset 2 Listen Only command
 |        numObjUpdate (IN) -- PGA460 can capture time-of-
flight, width, and amplitude for 1 to 8 objects.
 |              TCI is limited to time-of-flight measurement data only.
 |
 |  Returns:  none
 *-------------------------------------------------------------------*/
void pga460::ultrasonicCmd(byte cmd, byte numObjUpdate)
{
    numObj = numObjUpdate; // number of objects to detect
    byte bufCmd[4] = {syncByte, 0xFF, numObj, 0xFF}; // prepare bufCmd with 0xFF placeholders

    if (comm!=1)
    {
        memset(objTime, 0xFF, 8); // reset and idle-high TCI object buffer
    }

    switch (cmd)
    {
        case 0: // Send Preset 1 Burst + Listen command
        {
            bufCmd[1] = P1BL;
            bufCmd[3] = calcChecksum(P1BL);
            break;
        }
        case 1: // Send Preset 2 Burst + Listen command
        {
            bufCmd[1] = P2BL;
            bufCmd[3] = calcChecksum(P2BL);
            break;
        }
        case 3: // Send Preset 1 Listen Only command
        {
            bufCmd[1] = P1LO;
            bufCmd[3] = calcChecksum(P1LO);
            break;
        }
        case 4: // Send Preset 2 Listen Only command
        {
            bufCmd[1] = P2LO;
            bufCmd[3] = calcChecksum(P2LO);
            break;
        }
        default: return;
    }

    if(comm!=1)
    {
        Serial1.write(bufCmd, sizeof(bufCmd)); // serial transmit master data to initiate burst
```

```
and/or listen command
    }
    else
    {
        tciCommand(cmd); // send preset 1 or 2 burst-and-listen or listen-only command
        pga460::tciRecord(numObj); // log up to eight TCI object toggles
    }

    delay(100); // maximum record length is 65ms, so delay with margin
    return;
}

/*------------------------------------------------- pullUltrasonicMeasResult -----
 |  Function pullUltrasonicMeasResult
 |
 |  Purpose:  Read the ultrasonic measurement result data based on the last busrt and/or listen
command issued.
 |    Only applicable to UART and OWU modes.
 |
 |  Parameters:
 |        busDemo (IN) --
 When true, do not print error message for a failed reading when running bus demo
 |
 |  Returns:  If measurement data successfully read, return true.
 *-------------------------------------------------------------------*/
bool pga460::pullUltrasonicMeasResult(bool busDemo)
{
    if (comm != 1) // UART or OWU mode
    {
        pga460SerialFlush();

        memset(ultraMeasResult, 0, sizeof(ultraMeasResult));

        if (comm == 2)
        {
           owuShift = 2; // OWU receive buffer offset to ignore transmitted data
        }

        byte buf[3] = {syncByte, UMR, calcChecksum(UMR)};
        Serial1.write(buf, sizeof(buf)); //serial transmit master data to read ultrasonic
measurement results

        starttime = millis();
        while ( (Serial1.available()<(5+owuShift)) && ((millis() -
 starttime) < MAX_MILLIS_TO_WAIT) )
        {
            // wait in this loop until we either get +5 bytes of data, or 0.25 seconds have gone
by
        }

        if((Serial1.available() < (5+owuShift)))
        {
            if (busDemo == false)
            {
                // the data didn't come in - handle the problem here
                Serial.println("ERROR - Did not receive measurement results!");
            }
            return false;
        }
        else
        {
            for(int n=0; n<((2+(numObj*4))+owuShift); n++)
            {
               ultraMeasResult[n] = Serial1.read();
               delay(1);
            }
            if (comm == 2) // OWU mode only
            {
                //rearrange array for OWU UMR results
                for(int n=0; n<(2+(numObj*4)); n++)
                {
                    ultraMeasResult[n+1] = ultraMeasResult[n+owuShift]; //element 0 skipped due
to no diagnostic field returned
                }
            }
        }
    }
    return true;
}

/*------------------------------------------------- printUltrasonicMeasResult -----
 |  Function printUltrasonicMeasResult
 |
 |  Purpose:  Converts time-of-flight readout to distance in meters.
 |        Width and amplitude data only available in UART or OWU mode.
 |
 |  Parameters:
 |        umr (IN) -- Ultrasonic measurement result look-up selector:
 |                Distance (m)    Width    Amplitude
 |                --------------------------------
 |            Obj1        0        1        2
 |            Obj2        3        4        5
 |            Obj3        6        7        8
 |            Obj4        9        10        11
 |            Obj5        12        13        14
 |            Obj6        15        16        17
 |            Obj7        18        19        20
 |            Obj8        21        22        23
 |
 |  Returns:  double representation of distance (m), width (us), or amplitude (8-bit)
 *-------------------------------------------------------------------*/
double pga460::printUltrasonicMeasResult(byte umr)
{
    int speedSound = 343; // 343 degC at room temperature
    double objReturn = 0;
    uint16_t objDist = 0;
    uint16_t objWidth = 0;
    uint16_t objAmp = 0;

    switch (umr)
    {
        case 0: //Obj1 Distance (m)
        {
            objDist = (ultraMeasResult[1]<<8) + ultraMeasResult[2];
            objReturn = objDist/2*0.000001*speedSound;
            break;
        }
        case 1: //Obj1 Width (us)
        {
            objWidth = ultraMeasResult[3];
            objReturn= objWidth * 16;
            break;
        }
        case 2: //Obj1 Peak Amplitude
        {
            objAmp = ultraMeasResult[4];
            objReturn= objAmp;
            break;
        }

        case 3: //Obj2 Distance (m)
```

```
            {
                objDist = (ultraMeasResult[5]<<8) + ultraMeasResult[6];
                objReturn = objDist/2*0.000001*speedSound;
                break;
            }
            case 4: //Obj2 Width (us)
            {
                objWidth = ultraMeasResult[7];
                objReturn= objWidth * 16;
                break;
            }
            case 5: //Obj2 Peak Amplitude
            {
                objAmp = ultraMeasResult[8];
                objReturn= objAmp;
                break;
            }

            case 6: //Obj3 Distance (m)
            {
                objDist = (ultraMeasResult[9]<<8) + ultraMeasResult[10];
                objReturn = objDist/2*0.000001*speedSound;
                break;
            }
            case 7: //Obj3 Width (us)
            {
                objWidth = ultraMeasResult[11];
                objReturn= objWidth * 16;
                break;
            }
            case 8: //Obj3 Peak Amplitude
            {
                objAmp = ultraMeasResult[12];
                objReturn= objAmp;
                break;
            }
            case 9: //Obj4 Distance (m)
            {
                objDist = (ultraMeasResult[13]<<8) + ultraMeasResult[14];
                objReturn = objDist/2*0.000001*speedSound;
                break;
            }
            case 10: //Obj4 Width (us)
            {
                objWidth = ultraMeasResult[15];
                objReturn= objWidth * 16;
                break;
            }
            case 11: //Obj4 Peak Amplitude
            {
                objAmp = ultraMeasResult[16];
                objReturn= objAmp;
                break;
            }
            case 12: //Obj5 Distance (m)
            {
                objDist = (ultraMeasResult[17]<<8) + ultraMeasResult[18];
                objReturn = objDist/2*0.000001*speedSound;
                break;
            }
            case 13: //Obj5 Width (us)
            {
                objWidth = ultraMeasResult[19];
                objReturn= objWidth * 16;
                break;
            }
```

```
        case 14: //Obj5 Peak Amplitude
        {
            objAmp = ultraMeasResult[20];
            objReturn= objAmp;
            break;
        }
        case 15: //Obj6 Distance (m)
        {
            objDist = (ultraMeasResult[21]<<8) + ultraMeasResult[22];
            objReturn = objDist/2*0.000001*speedSound;
            break;
        }
        case 16: //Obj6 Width (us)
        {
            objWidth = ultraMeasResult[23];
            objReturn= objWidth * 16;
            break;
        }
        case 17: //Obj6 Peak Amplitude
        {
            objAmp = ultraMeasResult[24];
            objReturn= objAmp;
            break;
        }
        case 18: //Obj7 Distance (m)
        {
            objDist = (ultraMeasResult[25]<<8) + ultraMeasResult[26];
            objReturn = objDist/2*0.000001*speedSound;
            break;
        }
        case 19: //Obj7 Width (us)
        {
            objWidth = ultraMeasResult[27];
            objReturn= objWidth * 16;
            break;
        }
        case 20: //Obj7 Peak Amplitude
        {
            objAmp = ultraMeasResult[28];
            objReturn= objAmp;
            break;
        }
        case 21: //Obj8 Distance (m)
        {
            objDist = (ultraMeasResult[29]<<8) + ultraMeasResult[30];
            objReturn = objDist/2*0.000001*speedSound;
            break;
        }
        case 22: //Obj8 Width (us)
        {
            objWidth = ultraMeasResult[31];
            objReturn= objWidth * 16;
            break;
        }
        case 23: //Obj8 Peak Amplitude
        {
            objAmp = ultraMeasResult[32];
            objReturn= objAmp;
            break;
        }
        default: Serial.println("ERROR - Invalid object result!"); break;
    }
    return objReturn;
}

/*------------------------------------------------- runEchoDataDump -----
```

```
 |  Function runEchoDataDump
 |
 |  Purpose:  Runs a preset 1 or 2 burst and or listen command to capture 128 bytes of echo data
dump.
 |        Toggle echo data dump enable bit to enable/disable echo data dump mode.
 |
 |  Parameters:
 |        preset (IN) -- determines which preset command is run:
 |            • 0 = Preset 1 Burst + Listen command
 |            • 1 = Preset 2 Burst + Listen command
 |            • 2 = Preset 1 Listen Only command
 |            • 3 = Preset 2 Listen Only command
 |
 |  Returns:  none
 *-------------------------------------------------------------------*/
void pga460::runEchoDataDump(byte preset)
{
    if (comm != 1) // UART or OWU mode
    {
        pga460SerialFlush();

        // enable Echo Data Dump bit
        regAddr = 0x40;
        regData = 0x80;
        byte buf10[5] = {syncByte, SRW, regAddr, regData, calcChecksum(SRW)};
        Serial1.write(buf10, sizeof(buf10));
        delay(10);

        // run preset 1 or 2 burst and or listen command
        pga460::ultrasonicCmd(preset, 1);

        // disbale Echo Data Dump bit
        regData = 0x00;
        buf10[3] = regData;
        buf10[4] = calcChecksum(SRW);
        Serial1.write(buf10, sizeof(buf10));
    }
    else // TCI mode
    {
        EE_CNTRL = 0x80;         // enable echo data dump
        tciIndexRW(11, true);     // write to index 11

        delay(10);
        tciCommand(preset);     // run burst+listen command
        delay(100);                // delay for maximum record time length with margin

        EE_CNTRL = 0x00;         // disable echo data dump
        tciIndexRW(11, true);     // write to index 11
        delay(10);
    }
    return;
}

/*------------------------------------------------- pullEchoDataDump -----
 |  Function pullEchoDataDump
 |
 |  Purpose:  Read out 128 bytes of echo data dump (EDD) from latest burst and or listen command.
 |        For UART and OWU, readout individual echo data dump register values, instead in bulk.
 |        For TCI, perform index 12 read of all echo data dump values in bulk.
 |        TODO: Enable UART and OWU cmd7 transducer echo data dump bulk read.
 |
 |  Parameters:
 |        element (IN) -- element from the 128 byte EDD memory
 |
 |  Returns:  byte representation of EDD element value
 *-------------------------------------------------------------------*/
```

```
byte pga460::pullEchoDataDump(byte element)
{
    if (comm != 1) // UART or OWU mode
    {
        if (element == 0)
        {
            byte temp = 0;
            pga460SerialFlush();

            if (comm == 2)
            {
                owuShift = 4; // OWU receive buffer offset to ignore transmitted data
            }

            regAddr = 0x80; // start of EDD memory
            byte buf9[4] = {syncByte, SRR, regAddr, calcChecksum(SRR)};
            Serial1.write(buf9, sizeof(buf9)); // read first byte of EDD memory
            for(int m=0; m<128; m++) // loop readout by iterating through EDD address range
            {
               buf9[2] = regAddr;
               buf9[3] = calcChecksum(SRR);
               Serial1.write(buf9, sizeof(buf9));
               delay(30);

               for(int n=0; n<(3+owuShift); n++)
               {
                   if(n==(1 + owuShift))
                   {
                        echoDataDump[m] = Serial1.read();
                   }
                   else
                   {
                       temp = Serial1.read();
                   }
               }
               regAddr++;
            }
        }
        return echoDataDump[element];
    }
    else // TCI
    {
        if (element == 0)
        {
            tciIndexRW(12, false); //only run when first calling this function to read out the
entire EDD to the receive buffer
            delay (500); // wait until EDD read out is completed with margin
        }
        delay(10);
        return bufRecv[element];
    }
}

/*------------------------------------------------- runDiagnostics -----
 |  Function runDiagnostics
 |
 |  Purpose:  Runs a burst+listen command to capture frequency, decay, and voltage diagnostic.
 |        Runs a listen-only command to capture noise level.
 |        Captures die temperature of PGA460 device.
 |        Converts raw diagnostics to comprehensive units
 |
 |  Parameters:
 |        run (IN) -- issue a preset 1 burst-and-listen command
 |        diag (IN) -- diagnostic value to return:
 |             • 0 = frequency diagnostic (kHz)
 |             • 1 = decay period diagnostic (us)
 |             • 2 = die temperature (degC)
 |             • 3 = noise level (8bit)
 |
 |  Returns:  double representation of last captured diagnostic
 *-------------------------------------------------------------------*/
double pga460::runDiagnostics(byte run, byte diag)
{
    double diagReturn = 0;
    pga460SerialFlush();
    int elementOffset = 0; //Only non-zero for OWU mode.
    int owuShiftSysDiag = 0; // Only non-zero for OWU mode.

    if (comm != 1) // UART and OWU
    {
        if (comm == 2)
        {
            owuShift = 2; // OWU receive buffer offset to ignore transmitted data //DEBUG was 2
            owuShiftSysDiag = 1; //DEBUG
        }

        if (run == 1) // issue  P2 burst+listen, and run system diagnostics command to get latest
results
        {
            // run burst+listen command at least once for proper diagnostic analysis
            pga460::ultrasonicCmd(0, 1);    // always run preset 1 (long distance) burst+listen
for 1 object for system diagnostic


            delay(100); // record time length maximum of 65ms, so add margin
            pga460SerialFlush();

            byte buf8[3] = {syncByte, SD, calcChecksum(SD)};
            Serial1.write(buf8, sizeof(buf8)); //serial transmit master data to read system
diagnostic results

            starttime = millis();
            while ( (Serial1.available()<(4+owuShift-owuShiftSysDiag)) && ((millis() -
 starttime) < MAX_MILLIS_TO_WAIT) )
            {
                // wait in this loop until we either get +4 bytes of data or 0.25 seconds have
gone by
            }
            if(Serial1.available() < (4+owuShift-owuShiftSysDiag))
            {
                // the data didn't come in - handle the problem here
                Serial.println("ERROR - Did not receive system diagnostics!");
            }
            else
            {
                for(int n=0; n<(4+owuShift-owuShiftSysDiag); n++)
                {
                   diagMeasResult[n] = Serial1.read();
                }
            }
        }

        if (diag == 2) //run temperature measurement
        {
            tempOrNoise = 0; // temp meas
            byte buf4[4] = {syncByte, TNLM, tempOrNoise, calcChecksum(TNLM)};
            Serial1.write(buf4, sizeof(buf4)); //serial transmit master data to run temp
measurement

            delay(10);
            pga460SerialFlush();
            delay(10);
            byte buf6[3] = {syncByte, TNLR, calcChecksum(TNLR)};
            Serial1.write(buf6, sizeof(buf6)); //serial transmit master data to read temperature
and noise results

            delay(100);
        }

        if (diag == 3) // run noise level meas
        {
            tempOrNoise = 1; // noise meas
            byte buf4[4] = {syncByte, TNLM, tempOrNoise, calcChecksum(TNLM)};
            Serial1.write(buf4, sizeof(buf4)); //serial transmit master data to run noise level
measurement (requires at least 8.2ms of post-delay)

            delay(10);
            pga460SerialFlush();
            delay(10);

            byte buf6[3] = {syncByte, TNLR, calcChecksum(TNLR)}; //serial transmit master data to
read temperature and noise results
            Serial1.write(buf6, sizeof(buf6));

            delay(100);
        }

        if (diag == 2 || diag == 3) // pull temp and noise level results
        {
            starttime = millis();
            while ( (Serial1.available()<(4+owuShift-owuShiftSysDiag)) && ((millis() -
 starttime) < MAX_MILLIS_TO_WAIT) )
            {
                // wait in this loop until we either get +4 bytes of data or 0.25 seconds have
gone by
            }

            if(Serial1.available() < (4+owuShift-owuShiftSysDiag))
            {
                // the data didn't come in - handle the problem here
                Serial.println("ERROR - Did not receive temp/noise!");
            }
            else
            {
                for(int n=0; n<(4+owuShift-owuShiftSysDiag); n++)
                {
                   tempNoiseMeasResult[n] = Serial1.read();
                }

            }
        }
        elementOffset = owuShift-owuShiftSysDiag; // OWU only

    }
    else //TCI
    {
        if (run == true)
        {
            delay(10);
            tciCommand(6); // run noise level measurement command
            delay(15);
            tciCommand(1); //run preset 2 burst+listen command
            delay(100);    // maximum record length is 65ms, so wait with margin


            tciIndexRW(1, false); //read index1
            delay(10);
            for(int n=1; n<4; n++)
            {
               diagMeasResult[n] = bufRecv[n-1];
            }
            tempNoiseMeasResult[2] = diagMeasResult[3]; //clone temperature result to element
2

            delay(10);
            tciCommand(5); // run temperature measurement command
            delay(10);

            tciIndexRW(0,false); //read index0
            delay(10);

            tempNoiseMeasResult[1] = bufRecv[0]; //store temp readout to element 1
        }
        elementOffset = 0; // no offset required fot TCI
    }

    delay(100);

    switch (diag)
    {
        case 0: // convert to transducer frequency in kHz
            {
                diagReturn = (1 / (diagMeasResult[1+elementOffset] * 0.0000005)) / 1000;
            }
            break;
        case 1: // convert to decay period time in us
            {
                diagReturn = diagMeasResult[2+elementOffset] * 16;
            }
            break;
        case 2: //convert to temperature in degC
            {
                diagReturn = (tempNoiseMeasResult[1+elementOffset] - 64) / 1.5;
            }
            break;
        case 3: //noise floor level
            {
                diagReturn = tempNoiseMeasResult[2+elementOffset];
            }
            break;
        default: break;
    }

    return diagReturn;
}

/*------------------------------------------------- burnEEPROM -----
 |  Function burnEEPROM
 |
 |  Purpose:  Burns the EEPROM to preserve the working/shadow register values to EEPROM after
power
 |        cycling the PGA460 device. Returns EE_PGRM_OK bit to determine if EEPROM burn was
successful.
 |
 |  Parameters:
 |        none
 |
 |  Returns:  bool representation of EEPROM program success
 *-------------------------------------------------------------------*/
bool pga460::burnEEPROM()
{
    byte burnStat = 0;
    byte temp = 0;
    bool burnSuccess = false;

    if (comm != 1)
    {

        // Write "0xD" to EE_UNLCK to unlock EEPROM, and '0' to EEPRGM bit at EE_CNTRL register
        regAddr = 0x40; //EE_CNTRL
        regData = 0x68;
        byte buf10[5] = {syncByte, SRW, regAddr, regData, calcChecksum(SRW)};
        Serial1.write(buf10, sizeof(buf10));
        delay(1);

        // Write "0xD" to EE_UNLCK to unlock EEPROM, and '1' to EEPRGM bit at EE_CNTRL register
        regAddr = 0x40; //EE_CNTRL
        regData = 0x69;
        buf10[2] = regAddr;
        buf10[3] = regData;
        buf10[4] = calcChecksum(SRW);
        Serial1.write(buf10, sizeof(buf10));
        delay(1000);


        // Read back EEPROM program status
        if (comm == 2)
        {
            owuShift = 1; // OWU receive buffer offset to ignore transmitted data
        }
        pga460SerialFlush();
        regAddr = 0x40; //EE_CNTRL
        byte buf9[4] = {syncByte, SRR, regAddr, calcChecksum(SRR)};
        Serial1.write(buf9, sizeof(buf9));
        delay(10);
        for(int n=0; n<3; n++)
        {
           if(n==1-owuShift)
           {
                burnStat = Serial1.read(); // store EE_CNTRL data
           }
           else
           {
               temp = Serial1.read();
           }
        }
    }
    else
    {
        EE_CNTRL = 0x68;
        tciIndexRW(11, true);     // write to index 11 to EE_UNLCK to unlock EEPROM, and '0' to
EEPRGM bit at EE_CNTRL register
        delay(1);                 // immediately send the same UART or TCI command with the
EEPRGM bit set to '1'.
        EE_CNTRL = 0x69;
        tciIndexRW(11, true);     // write to index 11 to EE_UNLCK to unlock EEPROM, and '1' to
EEPRGM bit at EE_CNTRL register
        delay(1000);
        tciIndexRW(11, false);    // read back index 11 to review EE_PGRM_OK bit
        burnStat = bufRecv[0];
    }

    if((burnStat & 0x04) == 0x04){burnSuccess = true;} // check if EE_PGRM_OK bit is '1'

    return burnSuccess;
}

/*------------------------------------------------- broadcast -----
```

```
 |  Function broadcast
 |
 |  Purpose:  Send a broadcast command to bulk write the user EEPROM, TVG, and/or Threshold
values for all devices, regardless of UART_ADDR.
 |        Placehold for user EEPROM broadcast available. Note, all devices will update to the
same UART_ADDR in user EEPROM broadcast command.
 |        This function is not applicable to TCI mode.
 |
 |  Parameters:
 |        eeBulk (IN) -- if true, broadcast user EEPROM
 |        tvgBulk (IN) -- if true, broadcast TVG
 |        thrBulk (IN) -- if true, broadcast Threshold
 |
 |  Returns: none
 *-------------------------------------------------------------------*/
void pga460::broadcast(bool eeBulk, bool tvgBulk, bool thrBulk)
{

    // TVG broadcast command:
    if (tvgBulk == true)
    {
        byte buf24[10] = {syncByte, BC_TVGBW, TVGAIN0, TVGAIN1, TVGAIN2, TVGAIN3, TVGAIN4,
TVGAIN5, TVGAIN6, calcChecksum(BC_TVGBW)};
        Serial1.write(buf24, sizeof(buf24));
        delay(10);
    }

    // Threshold broadcast command:
    if (thrBulk == true)
    {
        byte buf25[35] = {syncByte, BC_THRBW, P1_THR_0, P1_THR_1, P1_THR_2, P1_THR_3, P1_THR_4,
P1_THR_5, P1_THR_6,
          P1_THR_7, P1_THR_8, P1_THR_9, P1_THR_10, P1_THR_11, P1_THR_12, P1_THR_13, P1_THR_14,
P1_THR_15,
          P2_THR_0, P2_THR_1, P2_THR_2, P2_THR_3, P2_THR_4, P2_THR_5, P2_THR_6,
          P2_THR_7, P2_THR_8, P2_THR_9, P2_THR_10, P2_THR_11, P2_THR_12, P2_THR_13, P2_THR_14,
P2_THR_15,
          calcChecksum(BC_THRBW)};

        Serial1.write(buf25, sizeof(buf25));
        delay(10);
    }

    // User EEPROM broadcast command (placeholder):
    if (eeBulk == true)
    {
        byte buf23[46] = {syncByte, BC_EEBW, USER_DATA1, USER_DATA2, USER_DATA3, USER_DATA4,
USER_DATA5, USER_DATA6,
            USER_DATA7, USER_DATA8, USER_DATA9, USER_DATA10, USER_DATA11, USER_DATA12,
USER_DATA13, USER_DATA14,
            USER_DATA15,USER_DATA16,USER_DATA17,USER_DATA18,USER_DATA19,USER_DATA20,
            TVGAIN0,TVGAIN1,TVGAIN2,TVGAIN3,TVGAIN4,TVGAIN5,TVGAIN6,INIT_GAIN,FREQUENCY,DEADTIME,

PULSE_P1,PULSE_P2,CURR_LIM_P1,CURR_LIM_P2,REC_LENGTH,FREQ_DIAG,SAT_FDIAG_TH,FVOLT_DEC,DECPL_TEMP,
            DSP_SCALE,TEMP_TRIM,P1_GAIN_CTRL,P2_GAIN_CTRL,calcChecksum(BC_EEBW)};

        Serial1.write(buf23, sizeof(buf23));
        delay(50);
    }

    return;
}


/*------------------------------------------------- calcChecksum -----
 |  Function calcChecksum
 |
 |  Purpose:  Calculates the UART checksum value based on the selected command and the user
EERPOM values associated with the command
 |        This function is not applicable to TCI mode.
 |
 |  Parameters:
 |        cmd (IN) -- the UART command for which the checksum should be calculated for
 |
 |  Returns: byte representation of calculated checksum value
 *-------------------------------------------------------------------*/
byte pga460::calcChecksum(byte cmd)
{
    int checksumLoops = 0;

    cmd = cmd & 0x001F; // zero-mask command address of cmd to select correct switch-
case statement

    switch(cmd)
    {
        case 0 : //P1BL
        case 1 : //P2BL
        case 2 : //P1LO
        case 3 : //P2LO
        case 17 : //BC_P1BL
        case 18 : //BC_P2BL
        case 19 : //BC_P1LO
        case 20 : //BC_P2LO
            ChecksumInput[0] = cmd;
            ChecksumInput[1] = numObj;
            checksumLoops = 2;
        break;
        case 4 : //TNLM
        case 21 : //TNLM
            ChecksumInput[0] = cmd;
            ChecksumInput[1] = tempOrNoise;
            checksumLoops = 2;
        break;
        case 5 : //UMR
        case 6 : //TNLR
        case 7 : //TEDD
        case 8 : //SD
        case 11 : //EEBR
        case 13 : //TVGBR
        case 15 : //THRBR
            ChecksumInput[0] = cmd;
            checksumLoops = 1;
        break;
        case 9 : //RR
            ChecksumInput[0] = cmd;
            ChecksumInput[1] = regAddr;
            checksumLoops = 2;
        break;
        case 10 : //RW
        case 22 : //BC_RW
            ChecksumInput[0] = cmd;
            ChecksumInput[1] = regAddr;
            ChecksumInput[2] = regData;
            checksumLoops = 3;
        break;
        case 14 : //TVGBW
        case 24 : //BC_TVGBW
            ChecksumInput[0] = cmd;
            ChecksumInput[1] = TVGAIN0;
            ChecksumInput[2] = TVGAIN1;
            ChecksumInput[3] = TVGAIN2;
            ChecksumInput[4] = TVGAIN3;
            ChecksumInput[5] = TVGAIN4;
            ChecksumInput[6] = TVGAIN5;
            ChecksumInput[7] = TVGAIN6;
            checksumLoops = 8;
        break;
        case 16 : //THRBW
        case 25 : //BC_THRBW
            ChecksumInput[0] = cmd;
            ChecksumInput[1] = P1_THR_0;
            ChecksumInput[2] = P1_THR_1;
            ChecksumInput[3] = P1_THR_2;
            ChecksumInput[4] = P1_THR_3;
            ChecksumInput[5] = P1_THR_4;
            ChecksumInput[6] = P1_THR_5;
            ChecksumInput[7] = P1_THR_6;
            ChecksumInput[8] = P1_THR_7;
            ChecksumInput[9] = P1_THR_8;
            ChecksumInput[10] = P1_THR_9;
            ChecksumInput[11] = P1_THR_10;
            ChecksumInput[12] = P1_THR_11;
            ChecksumInput[13] = P1_THR_12;
            ChecksumInput[14] = P1_THR_13;
            ChecksumInput[15] = P1_THR_14;
            ChecksumInput[16] = P1_THR_15;
            ChecksumInput[17] = P2_THR_0;
            ChecksumInput[18] = P2_THR_1;
            ChecksumInput[19] = P2_THR_2;
            ChecksumInput[20] = P2_THR_3;
            ChecksumInput[21] = P2_THR_4;
            ChecksumInput[22] = P2_THR_5;
            ChecksumInput[23] = P2_THR_6;
            ChecksumInput[24] = P2_THR_7;
            ChecksumInput[25] = P2_THR_8;
            ChecksumInput[26] = P2_THR_9;
            ChecksumInput[27] = P2_THR_10;
            ChecksumInput[28] = P2_THR_11;
            ChecksumInput[29] = P2_THR_12;
            ChecksumInput[30] = P2_THR_13;
            ChecksumInput[31] = P2_THR_14;
            ChecksumInput[32] = P2_THR_15;
            checksumLoops = 33;
        break;
        case 12 : //EEBW
        case 23 : //BC_EEBW
            ChecksumInput[0] = cmd;
            ChecksumInput[1] = USER_DATA1;
            ChecksumInput[2] = USER_DATA2;
            ChecksumInput[3] = USER_DATA3;
            ChecksumInput[4] = USER_DATA4;
            ChecksumInput[5] = USER_DATA5;
            ChecksumInput[6] = USER_DATA6;
            ChecksumInput[7] = USER_DATA7;
            ChecksumInput[8] = USER_DATA8;
            ChecksumInput[9] = USER_DATA9;
            ChecksumInput[10] = USER_DATA10;
            ChecksumInput[11] = USER_DATA11;
            ChecksumInput[12] = USER_DATA12;
            ChecksumInput[13] = USER_DATA13;
            ChecksumInput[14] = USER_DATA14;
            ChecksumInput[15] = USER_DATA15;
            ChecksumInput[16] = USER_DATA16;
            ChecksumInput[17] = USER_DATA17;
            ChecksumInput[18] = USER_DATA18;
            ChecksumInput[19] = USER_DATA19;
            ChecksumInput[20] = USER_DATA20;
            ChecksumInput[21] = TVGAIN0;
            ChecksumInput[22] = TVGAIN1;
            ChecksumInput[23] = TVGAIN2;
            ChecksumInput[24] = TVGAIN3;
            ChecksumInput[25] = TVGAIN4;
            ChecksumInput[26] = TVGAIN5;
            ChecksumInput[27] = TVGAIN6;
            ChecksumInput[28] = INIT_GAIN;
            ChecksumInput[29] = FREQUENCY;
            ChecksumInput[30] = DEADTIME;
            ChecksumInput[31] = PULSE_P1;
            ChecksumInput[32] = PULSE_P2;
            ChecksumInput[33] = CURR_LIM_P1;
            ChecksumInput[34] = CURR_LIM_P2;
            ChecksumInput[35] = REC_LENGTH;
            ChecksumInput[36] = FREQ_DIAG;
            ChecksumInput[37] = SAT_FDIAG_TH;
            ChecksumInput[38] = FVOLT_DEC;
            ChecksumInput[39] = DECPL_TEMP;
            ChecksumInput[40] = DSP_SCALE;
            ChecksumInput[41] = TEMP_TRIM;
            ChecksumInput[42] = P1_GAIN_CTRL;
            ChecksumInput[43] = P2_GAIN_CTRL;
            checksumLoops = 44;
        break;
        default: break;
    }

    if (ChecksumInput[0]<17) //only re-append command address for non-broadcast commands.
    {
        ChecksumInput[0] = ChecksumInput[0] + (uartAddr << 5);
    }

    uint16_t carry = 0;

    for (int i = 0; i < checksumLoops; i++)
    {
        if ((ChecksumInput[i] + carry) < carry)
        {
            carry = carry + ChecksumInput[i] + 1;
        }
        else
        {
            carry = carry + ChecksumInput[i];
        }

        if (carry > 0xFF)
        {
          carry = carry - 255;
        }
    }

    carry = (~carry & 0x00FF);
    return carry;
}

/*------------------------------------------------- tciIndexRW -----
 |  Function tciIndexRW
 |
 |  Purpose:  Read or write the TCI index command.
 |        TODO: Enable all commands to be written. Update user EEPROM variables based on index
read.
 |
 |  Parameters:
 |        index (IN) -- TCI index (0-15) to read or write.
 |        wTrue (IN) --
 when true, issue a TCI write command. When false, issue a TCI read command.
 |
 |  Returns: none
 *-------------------------------------------------------------------*/
void pga460::tciIndexRW(byte index, bool wTrue)
{
    int dataLength = 0;        // number of bits per TCI index
    String zeroString = "";    // string of zeros to append to the end of the binary string for
the checksum calculation
    String dataString = "";    // entire index data string with appended zeros for checksum
calculation
    byte dataLoops = 0;        // based on the number elements to be passed into the checksum
calaculation after appending zeros
    byte bufTCI[46];        // transmit TCI buffer for all index commands
    byte data = 0xFF;        // idle-high data transmit data
    byte zeroPadding = 0;    // byte-
number of zeros to append to the end of the binary string for the checksum calculation
    byte bitIgnore = 0;        // number of bits to ignore at the end of the concatenated bufTCI
index string

    if (wTrue == true) // TCI write command
    {
        bufTCI[0] = 0x1F & (0x10 + index); // set first byte with write bit and index
        switch(index)
        {
            case 0: dataLength = 8; break; //read only
            case 1: dataLength = 24; break; //read only
            case 2: zeroPadding = 3; dataLength = 8; zeroString = "000";  bitIgnore = 0;
                bufTCI[1] = FREQUENCY;
                break;
            case 3: zeroPadding = 1; dataLength = 18; zeroString = "0"; bitIgnore = 0;
                //TODO
                break;
            case 4: zeroPadding = 3; dataLength = 8; zeroString = "000"; bitIgnore = 0;
                //TODO
                break;
            case 5: zeroPadding = 3; dataLength = 124; zeroString = "000"; bitIgnore = 4;
                    bufTCI[1] = P1_THR_0;
                    bufTCI[2] = P1_THR_1;
                    bufTCI[3] = P1_THR_2;
                    bufTCI[4] = P1_THR_3;
                    bufTCI[5] = P1_THR_4;
                    bufTCI[6] = P1_THR_5;
                    bufTCI[7] = P1_THR_6;
                    bufTCI[8] = P1_THR_7;
                    bufTCI[9] = P1_THR_8;
                    bufTCI[10] = P1_THR_9;
                    bufTCI[11] = P1_THR_10;
                    bufTCI[12] = P1_THR_11;
                    bufTCI[13] = P1_THR_12;
                    bufTCI[14] = P1_THR_13;
                    bufTCI[15] = P1_THR_14;
                    bufTCI[16] = (P1_THR_15 & 0x0F) << 4; //TH_P1_OFF only
                break;
            case 6: zeroPadding = 3; dataLength = 124; zeroString = "000"; bitIgnore = 4;
                    bufTCI[1] = P2_THR_0;
                    bufTCI[2] = P2_THR_1;
                    bufTCI[3] = P2_THR_2;
                    bufTCI[4] = P2_THR_3;
                    bufTCI[5] = P2_THR_4;
                    bufTCI[6] = P2_THR_5;
                    bufTCI[7] = P2_THR_6;
                    bufTCI[8] = P2_THR_7;
                    bufTCI[9] = P2_THR_8;
                    bufTCI[10] = P2_THR_9;
                    bufTCI[11] = P2_THR_10;
                    bufTCI[12] = P2_THR_11;
                    bufTCI[13] = P2_THR_12;
                    bufTCI[14] = P2_THR_13;
                    bufTCI[15] = P2_THR_14;
                    bufTCI[16] = (P2_THR_15 & 0x0F) << 4; //TH_P2_OFF only
                break;
            case 7: zeroPadding = 1; dataLength = 42; zeroString = "0"; bitIgnore = 0;
                //TODO
                break;
            case 8: zeroPadding = 3; dataLength = 56; zeroString = "000"; bitIgnore = 0;
                    bufTCI[1] = TVGAIN0;
                    bufTCI[2] = TVGAIN1;
                    bufTCI[3] = TVGAIN2;
                    bufTCI[4] = TVGAIN3;
                    bufTCI[5] = TVGAIN4;
                    bufTCI[6] = TVGAIN5;
                    bufTCI[7] = TVGAIN6;
                break;
            case 9: zeroPadding = 3; dataLength = 160; zeroString = "000"; bitIgnore = 0;
                //TODO
                break;
            case 10: zeroPadding = 5; dataLength = 46; zeroString = "00000"; bitIgnore = 0;
                //TODO
                break;
            case 11: zeroPadding = 3; dataLength = 8; zeroString = "000"; bitIgnore = 0;
                bufTCI[1] = EE_CNTRL;
                break;
            case 12: dataLength = 1024; break; //read only
            case 13: zeroPadding = 3; zeroString = "000";  dataLength = 352; bitIgnore = 0;

                    bufTCI[1] = USER_DATA1;
                    bufTCI[2] = USER_DATA2;
                    bufTCI[3] = USER_DATA3;
                    bufTCI[4] = USER_DATA4;
                    bufTCI[5] = USER_DATA5;
                    bufTCI[6] = USER_DATA6;
                    bufTCI[7] = USER_DATA7;
                    bufTCI[8] = USER_DATA8;
                    bufTCI[9] = USER_DATA9;
                    bufTCI[10] = USER_DATA10;
                    bufTCI[11] = USER_DATA11;
                    bufTCI[12] = USER_DATA12;
                    bufTCI[13] = USER_DATA13;
                    bufTCI[14] = USER_DATA14;
                    bufTCI[15] = USER_DATA15;
                    bufTCI[16] = USER_DATA16;
                    bufTCI[17] = USER_DATA17;
                    bufTCI[18] = USER_DATA18;
                    bufTCI[19] = USER_DATA19;
                    bufTCI[20] = USER_DATA20;
                    bufTCI[21] = TVGAIN0;
                    bufTCI[22] = TVGAIN1;
                    bufTCI[23] = TVGAIN2;
                    bufTCI[24] = TVGAIN3;
                    bufTCI[25] = TVGAIN4;
                    bufTCI[26] = TVGAIN5;
                    bufTCI[27] = TVGAIN6;
                    bufTCI[28] = INIT_GAIN;
                    bufTCI[29] = FREQUENCY;
                    bufTCI[30] = DEADTIME;
                    bufTCI[31] = PULSE_P1;
                    bufTCI[32] = PULSE_P2;
                    bufTCI[33] = CURR_LIM_P1;
                    bufTCI[34] = CURR_LIM_P2;
                    bufTCI[35] = REC_LENGTH;
                    bufTCI[36] = FREQ_DIAG;
                    bufTCI[37] = SAT_FDIAG_TH;
```

```
                    bufTCI[38] = FVOLT_DEC;
                    bufTCI[39] = DECPL_TEMP;
                    bufTCI[40] = DSP_SCALE;
                    bufTCI[41] = TEMP_TRIM;
                    bufTCI[42] = P1_GAIN_CTRL;
                    bufTCI[43] = P2_GAIN_CTRL;
                    bufTCI[44] = EE_CRC;
                break;
            case 14: break; //read only (reserved)
            case 15: dataLength = 16; break; //read only
            default: return;
        }

        // calculate checksum
            // convert byte to binary string
                dataLoops = ((dataLength+((zeroPadding+bitIgnore)-3))/8) + 1;

                String tempString = "";
                for (int i=0; i<dataLoops; i++)
                {
                    tempString = String((int)bufTCI[i],BIN);
                    while (tempString.length() < 8)
                    {
                        tempString = "0" + tempString;    // add leading zero to get 8 bit BIN
representaiton
                    }
                    dataString.concat(tempString);
                }
                dataString = dataString.substring(3); // truncate leading zeros
                dataString.concat(zeroString); // append zero padding to binary string

            // convert binary string to bytes for checksum calculation
                String parsed = "";
                byte value = 0;
                for(int k=0; k < dataLoops; k++)
                {
                    parsed = dataString.substring(k*8,(k*8)+8);
                    char s[9];
                    parsed.toCharArray(s,9);
                    for (int i=0; i< strlen(s); i++)  // for every character in the string
strlen(s) returns the length of a char array
                    {
                      value *= 2; // double the result so far
                      if (s[i] == '1') value++;  //add 1 if needed
                    }
                    bufTCI[k] = value;
                }

            // generate TCI checksum
            uint16_t carry = 0;
                for (int i = 0; i < dataLoops; i++)
                {
                    if ((bufTCI[i] + carry) < carry)
                    {
                        carry = carry + bufTCI[i] + 1;
                    }
                    else
                    {
                        carry = carry + bufTCI[i];
                    }

                    if (carry > 0xFF)
                    {
                      carry = carry - 255;
                    }
                }
                carry = (~carry & 0x00FF);

        // send CFG_TCI low pulse of 1.27ms
        tciCommand(4);

        // transmit r/w , index, and data bits.
            for (int m = 0; m < dataLoops-1; m++)
            {
                data = bufTCI[m];
                pga460::tciByteToggle(data,0); // send bits 7..0
            }

       // send last byte without zero padding
            data = bufTCI[dataLoops-1];
            {
                data = data >> (zeroPadding+bitIgnore);
                pga460::tciByteToggle(data,(zeroPadding+bitIgnore));
            }

       // send checksum
           data = (byte)carry;
           pga460::tciByteToggle(data,0);
    }

    else // TCI read command
    {
        int recvLength = 0;     // number of bits to expect for the index to be read
        bool recvState = 0xFF;    // receive state initiated to idle high
        bool lastState = 0xFF;    // last state read initiated to idle high
        int element = 0;        // bufRecv byte element to save bit capture to
        int bitCount = 0;        // number of bits read to auto increment bufRecv element after 8
hits

        switch (index)
        {
            case 0: recvLength = 8; break;
            case 1: recvLength = 24; break;
            case 2: recvLength = 8; break;
            case 3: recvLength = 18; break;
            case 4: recvLength = 8; break;
            case 5: recvLength = 124; break;
            case 6: recvLength = 124; break;
            case 7: recvLength = 42; break;
            case 8: recvLength = 56; break;
            case 9: recvLength = 160; break;
            case 10: recvLength = 46; break;
            case 11: recvLength = 8; break;
            case 12: recvLength = 1024; break;
            case 13: recvLength = 352; break;
            case 14: recvLength = 0; break;
            case 15: recvLength = 16; break;
            default: return;
        }

        memset(bufRecv, 0xFF, sizeof(bufRecv)); // idle-high receive buffer data
        starttime = millis();

        // send CFG_TCI low pulse of 1.27ms
        tciCommand(4);
        data = 0x1F & (0x00 + index);
        pga460::tciByteToggle(data,3);
        delayMicroseconds(100); //TCI deadtime

        // capture first response toggle by sampling center of 300us TCI bit indicate 0 or 1
        delayMicroseconds(150);
        lastState=digitalRead(TCI_RX);
        bitWrite(bufRecv[element], 7-bitCount, digitalRead(TCI_RX));
        bitCount++;

        while((millis() - starttime) < 500) // timeout after 0.5 seconds
        {
            recvState = digitalRead(TCI_RX);
            if (((recvState != lastState) && (recvState == 0))) // check for high-to-low toggle
            {
                // sample center of 300us TCI bit indicate 0 or 1
                delayMicroseconds(150);
                bitWrite(bufRecv[element], 7-bitCount, digitalRead(TCI_RX));
                bitCount++;
                if (bitCount == 8)
                {
                    bitCount = 0;
                    element++;
                }
            }
            lastState = recvState;
            delayMicroseconds(10); // master defined deglitcher timeout
        }
    }
    return;
}

/*------------------------------------------------- tciByteToggle -----
 |  Function tciByteToggle
 |
 |  Purpose:  Toggle the TCI_TX pin based on the bit data of the byte data passed in.
 |        A bit value of '1' toggles TCI_TX low for 100us, then holds it high for 200us.
 |        A bit value of '0' toggles TCI_TX low for 200us, then holds it high for 100us.
 |
 |  Parameters:
 |        data (IN) -- byte value to bit parse.
 |        zeroPadding (IN) --
 bit toggle based on the number of zeros padded. Zero padding is for checksum calculation only.
 |
 |  Returns: none
 *-------------------------------------------------------------------*/
void pga460::tciByteToggle(byte data, byte zeroPadding)
{
    byte mask = 0x80;
    int numBits = 8;
    switch (zeroPadding)
    {
        case 0: mask = 0x80; numBits = 8; break;
        case 1: mask = 0x40; numBits = 7; break;
        case 2: mask = 0x20; numBits = 6; break;
        case 3: mask = 0x10; numBits = 5; break;
        case 4: mask = 0x08; numBits = 4; break;
        case 5: mask = 0x04; numBits = 3; break;
        case 6: mask = 0x02; numBits = 2; break;
        case 7: mask = 0x01; numBits = 1; break;
        default: return;
    }

    for (int n = 0; n < numBits; n++)
       {
           // set line low for 100us if bit is 1, low for 200us if bit is 0
           if (data & mask) // consider leftmost bit (MSB out first)
           {
            digitalWrite(TCI_TX, LOW);
            delayMicroseconds(100);
            digitalWrite(TCI_TX, HIGH);
            delayMicroseconds(200);
           }
           else
           {
            digitalWrite(TCI_TX, LOW);
            delayMicroseconds(200);
            digitalWrite(TCI_TX, HIGH);
            delayMicroseconds(100);
           }
           data <<= 1; // shift byte left so next bit will be leftmost
       }
       return;
}

/*------------------------------------------------- tciRecord -----
 |  Function tciRecord
 |
 |  Purpose:  Record TCI_RX toggle burst and/or low activity to time stamp high-to-
low transitions representing
        time-of-flight measurements. The time-of-
flight is captures in microseconds, and saved to the ultrasonic
        measurement results array to later convert time-of-flight to distance in meters.
 |
 |  Parameters:
 |        data (IN) -- byte value to bit parse.
 |        numObj (IN) --
 number of objects/toggles to monitor the TCI_RX line for (limited to 8 for this library)
 |
 |  Returns: none
 *-------------------------------------------------------------------*/
void pga460::tciRecord(byte numObj)
{
    bool recvState = 0;
    bool lastState = 0;
    tciToggle = micros();
    byte objCount = 0;
    starttime = millis();
    delayMicroseconds(300); //wait until after STAT bits are toggled by PGA460

    while(((millis() -
 starttime) < 100) && (objCount < numObj)) // timeout after 100ms, or after set number of objects
are registered
    {
        recvState = digitalRead(TCI_RX);
        if (((recvState != lastState) && (recvState == 0))) // check for high-to-
low toggle of TCI_RX line
        {
            objTime[objCount] = (int)(micros() - tciToggle); // capture time-of-flight
            objCount++;
        }
        lastState = recvState;
        delayMicroseconds(10); // master implemented deglitcher //8cm resolution due to micros
timer
    }

    if (objCount == (numObj-1)) // if number of objects fills before timer expires
    {
        delay(100-(millis() - starttime)); //wait a total time of 100ms regardless
    }

    // save each TCI time-of-
flight to ultrasonic measurement results array (16 bit parsed into two 8 byte elements)
    for (int i = 0; i < objCount; i++)
    {
        ultraMeasResult[(i*4)+1] = (objTime[i] >> 8) & 0x00FF; // MSB
        ultraMeasResult[(i*4)+2] = 0x00FF; //LSB
    }
}
```

```
/*-------------------------------------------------- tciCommand -----
 |  Function tciCommand
 |
 |  Purpose:  Toggle TCI_TX low for micro second duration based on nominal requirement of TCI
command.
 |
 |  Parameters:
 |        cmd (IN) -- which TCI command to issue
 |             • 0 = BURST/LISTEN (Preset1)
 |             • 1 = BURST/LISTEN (Preset2)
 |             • 2 = LISTEN only (Preset1)
 |             • 3 = LISTEN only (Preset2)
 |             • 4 = Device configuration
 |             • 5 = Temperature measurement
 |             • 6 = Noise level
 |
 |  Returns: none
 *-------------------------------------------------------------------*/
void pga460::tciCommand(byte cmd)
{
    digitalWrite(TCI_TX, LOW);

    switch (cmd)
    {
        case 0: delayMicroseconds(400); break; //send P1BL_TCI low pulse
        case 1: delayMicroseconds(1010); break; //send P2BL_TCI low pulse
        case 2: delayMicroseconds(780); break; //send P1LO_TCI low pulse
        case 3: delayMicroseconds(580); break; //send P2LO_TCI low pulse
        case 4: delayMicroseconds(1270); break; //send CFG_TCI low pulse
        case 5: delayMicroseconds(1550); break; //send TEMP_TCI low pulse
        case 6: delayMicroseconds(2200); break; //send NOISE_TCI low pulse
        default: break;
    }

    digitalWrite(TCI_TX, HIGH);
    delayMicroseconds(100); //TCI deadtime
}

/*-------------------------------------------------- pga460SerialFlush -----
 |  Function pga460SerialFlush
 |
 |  Purpose:  Clears the MSP430's UART receiver buffer
 |
 |  Parameters:
 |        none
 |
 |  Returns: none
 *-------------------------------------------------------------------*/
void pga460::pga460SerialFlush()
{
    delay(10);
    Serial1.flush();
    while((Serial1.available() > 0))// || (Serial1.read() < 0xFF))
    {
        char temp = Serial1.read();
        //Serial1.flush();
    }

    //redundant clear
    for (int i = 0; i < 10; i++)
     {
       while (Serial.available() > 0)
       {
         char k = Serial.read();
         delay(1);
```

```
            }
            delay(1);
          }

        Serial1.flush();
        return;
    }

    /*------------------------------------------------ toggleLEDs -----
     |  Function toggleLEDs
     |
     |  Purpose:  Set the BOOSTXL-PGA460 diagnostic LED state to ON or OFF.
     |
     |  Parameters:
     |        ds1State (IN) -- state of BOOSTXL-PGA460 RED LED populated at D9
     |        fdiagState (IN) -- state of BOOSTXL-PGA460 RED LED populated at D8
     |        vdiagate (IN) -- state of BOOSTXL-PGA460 RED LED populated at D7
     |
     |  Returns:  none
     *-------------------------------------------------------------------*/
    void pga460::toggleLEDs(bool ds1State, bool fdiagState, bool vdiagState)
    {
        digitalWrite(DS1_LED, ds1State); digitalWrite(F_DIAG_LED, fdiagState);
    digitalWrite(V_DIAG_LED, vdiagState);
        return;
    }
```

5.2 pga460_ussc.h

The *pga460_ussc.h* code is as follows:

```
/*
 * PGA460_USSC.h
 * Created by A. Whitehead <make@energia.nu>, Initial: Nov 2016, Updated: Aug 2017
 * Released into the public domain.
 * This is free software: you can redistribute it and/or modify
 * it under the terms of the GNU Lesser General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 *
 * You should have received a copy of the GNU Lesser General Public License.
 * If not, see <http://www.gnu.org/licenses/>.
 *
 * Copyright 2016 A. Whitehead <make@energia.nu>
 * Updated: Aug 2017
 */

#include <Energia.h>
#include <string.h>

class pga460
{
  public:
    pga460();
    byte pullEchoDataDump(byte element);
    void initBoostXLPGA460(byte mode, uint32_t baud, byte uartAddrUpdate);
    void defaultPGA460(byte xdcr);
    void initThresholds(byte thr);
    void initTVG(byte agr, byte tvg);
    void ultrasonicCmd(byte cmd, byte numObjUpdate);
    void runEchoDataDump(byte preset);
    void broadcast(bool eeBulk, bool tvgBulk, bool thrBulk);
    void toggleLEDs(bool ds1State, bool fdiagState, bool vdiagState);
    bool burnEEPROM();
    bool pullUltrasonicMeasResult(bool busDemo);
```

```
        double printUltrasonicMeasResult(byte umr);
        double runDiagnostics(byte run, byte diag);

      private:
        byte calcChecksum(byte cmd);
        void pga460SerialFlush();
        void tciRecord(byte numObj);
        void tciByteToggle(byte data, byte zeroPadding);
        void tciIndexRW(byte index, bool write);
        void tciCommand(byte cmd);
    };
```

Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from Original (February 2017) to A Revision	Page
• Added expanded support to TCI and OWU interface modes.	1
• Deleted redundant code-snippets throughout entire document.	1
• Added setup description of OWU mode, and hardware requirements.	4
• Added setup description of TCI mode, and hardware requirements.	4
• Changed COM Terminal Mode screen capture based on version 1.0.3 of GetDistance.ino sketch.	9
• Changed the source code to version 1.0.3 of GetDistance.ino sketch.	10
• Changed the source code to version 1.0.3 of PGA460_USSC library files.	17

IMPORTANT NOTICE FOR TI DESIGN INFORMATION AND RESOURCES

Texas Instruments Incorporated ('TI") technical, application or other design advice, services or information, including, but not limited to, reference designs and materials relating to evaluation modules, (collectively, "TI Resources") are intended to assist designers who are developing applications that incorporate TI products; by downloading, accessing or using any particular TI Resource in any way, you (individually or, if you are acting on behalf of a company, your company) agree to use it solely for this purpose and subject to the terms of this Notice.

TI's provision of TI Resources does not expand or otherwise alter TI's applicable published warranties or warranty disclaimers for TI products, and no additional obligations or liabilities arise from TI providing such TI Resources. TI reserves the right to make corrections, enhancements, improvements and other changes to its TI Resources.

You understand and agree that you remain responsible for using your independent analysis, evaluation and judgment in designing your applications and that you have full and exclusive responsibility to assure the safety of your applications and compliance of your applications (and of all TI products used in or for your applications) with all applicable regulations, laws and other applicable requirements. You represent that, with respect to your applications, you have all the necessary expertise to create and implement safeguards that (1) anticipate dangerous consequences of failures, (2) monitor failures and their consequences, and (3) lessen the likelihood of failures that might cause harm and take appropriate actions. You agree that prior to using or distributing any applications that include TI products, you will thoroughly test such applications and the functionality of such TI products as used in such applications. TI has not conducted any testing other than that specifically described in the published documentation for a particular TI Resource.

You are authorized to use, copy and modify any individual TI Resource only in connection with the development of applications that include the TI product(s) identified in such TI Resource. NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE TO ANY OTHER TI INTELLECTUAL PROPERTY RIGHT, AND NO LICENSE TO ANY TECHNOLOGY OR INTELLECTUAL PROPERTY RIGHT OF TI OR ANY THIRD PARTY IS GRANTED HEREIN, including but not limited to any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information regarding or referencing third-party products or services does not constitute a license to use such products or services, or a warranty or endorsement thereof. Use of TI Resources may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

TI RESOURCES ARE PROVIDED "AS IS" AND WITH ALL FAULTS. TI DISCLAIMS ALL OTHER WARRANTIES OR REPRESENTATIONS, EXPRESS OR IMPLIED, REGARDING TI RESOURCES OR USE THEREOF, INCLUDING BUT NOT LIMITED TO ACCURACY OR COMPLETENESS, TITLE, ANY EPIDEMIC FAILURE WARRANTY AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

TI SHALL NOT BE LIABLE FOR AND SHALL NOT DEFEND OR INDEMNIFY YOU AGAINST ANY CLAIM, INCLUDING BUT NOT LIMITED TO ANY INFRINGEMENT CLAIM THAT RELATES TO OR IS BASED ON ANY COMBINATION OF PRODUCTS EVEN IF DESCRIBED IN TI RESOURCES OR OTHERWISE. IN NO EVENT SHALL TI BE LIABLE FOR ANY ACTUAL, DIRECT, SPECIAL, COLLATERAL, INDIRECT, PUNITIVE, INCIDENTAL, CONSEQUENTIAL OR EXEMPLARY DAMAGES IN CONNECTION WITH OR ARISING OUT OF TI RESOURCES OR USE THEREOF, AND REGARDLESS OF WHETHER TI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You agree to fully indemnify TI and its representatives against any damages, costs, losses, and/or liabilities arising out of your non-compliance with the terms and provisions of this Notice.

This Notice applies to TI Resources. Additional terms apply to the use and purchase of certain types of materials, TI products and services. These include; without limitation, TI's standard terms for semiconductor products http://www.ti.com/sc/docs/stdterms.htm), evaluation modules, and samples (http://www.ti.com/sc/docs/sampterms.htm).

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265