# A QEI Shield for Arduino

A common way to detect motor angle is to attach an encoder device to its shaft. An incremental encoder is often preferred to an absolute encoder due to less signal wirings, though it requires additional circuitry to convert quadrature signal pair A and B to numeric counts. Figure A.1 shows how an incremental encoder works. When the motor rotates, the slits on the disc pass and block the lights from LEDs to the receiving sensors, generating pulse A and B which are 90 degree out-of-phase. Some encoder also provides an index pulse Z each revolution, which is useful for some application such as homing of CNC machine axis. The A, B, and Z pulses are shown in Figure A.2. For all experiments in this book we consider only the A and B signals.

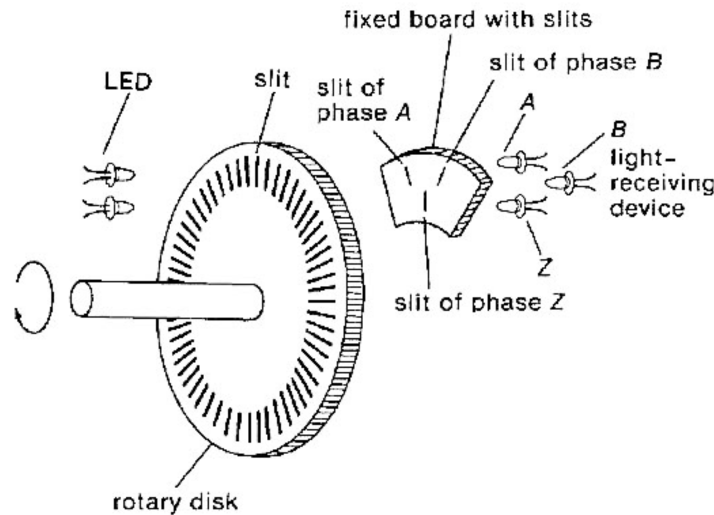


Figure A.1 physical structure of incremental encoder

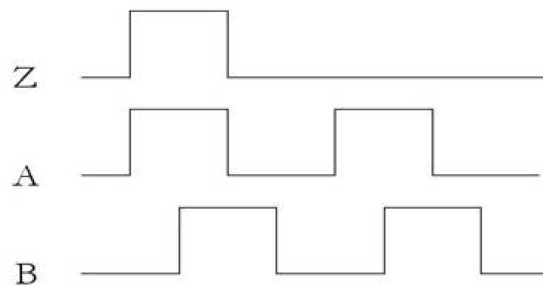


Figure A.2 the quadrature and index signals from an incremental encoder

Though it is possible to feed the A, B signals directly to digital pins of Arduino and write some software routine to read their states and convert to position counts, this approach is problematic when the motor speed increases. The sampling rate needs to keep up with the higher frequency of quadrature signals. Unless for very crude encoder resolution, eventually this high sampling rate requirement would exceed the limited performance of low-end Arduino board.

Better accuracy is achieved when hardware unit dedicated to encoder interface is used. There are commercial ASICs for such purpose, preferably with serial communication interface such as SPI and I2C to minimize the number of pins used. An example is LS7366, from LSI Computer Systems Inc., that has 32-bit counter and SPI hardware on the chip.

You may easily find an Arduino shield for encoder interface on the market. But, for the fun of it, we demonstrate how to create one from another microcontroller that has an quadrature encoder module and interface, in this case I2C is selected. One such μC is PIC24EP motor control family from Microchip. Here we choose PIC24EP128MC202 (In fact, the flash memory of 128K well exceeds the requirement. I just have one in my drawer. You can choose one with less flash memory.)

## A.1 Hardware Prototype

The shield is named QEII2C. Figure A.3 shows top and bottom view of the prototype. All components are soldered and wired on a blank protoboard designed to be used with Arduino. It may look somewhat too messy than it should be. Some jumpers, and the yellow push button switch, are installed for future development. For example, the user may select between I2C or SPI. To simplify the discussion, here we mention only I2C interface.

Consult Microchip datasheet on the basic schematics. To eliminate XTAL circuit, we use internal FRC to generate system clock for the μC. This works fine for a synchronous serial communication where the PIC acts as a slave device. Table A.1 summarizes the pin numbers and functions used. Trivial pins such as VDD, AVDD, VSS, AVSS, VCAP, MCLR are omitted.

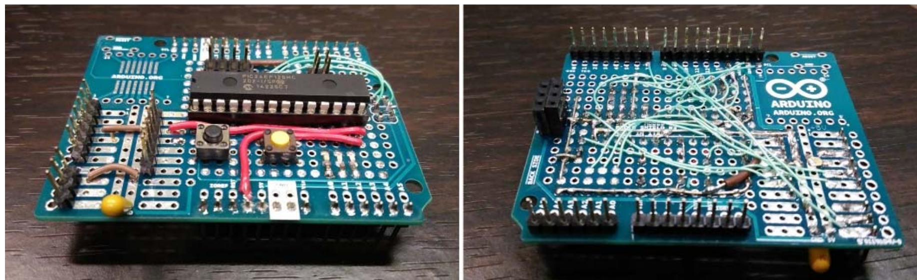


Figure A.3 top and bottom views of QEII2C shield

| Pin # | Type | Function | Description |
|---|---|---|---|
| 2 | O | MODE LED | Mode indicator* |
| 3 | O | RD LED | I2C read indicator |
| 4 | O | WR LED | I2C write indicator |
| 5 | I | MODE SW | Mode select* |
| 6 | I | PGEC1 | In-circuit program pin |
| 7 | I | PGED1 | In-circuit program pin |
| 14 | I2C | ASDA2 | I2C data |
| 15 | I2C | ASCK2 | I2C clock |
| 23 | I | QEA | Encoder A channel |
| 24 | I | QEB | Encoder B channel |

Table A.1 PIC24EP128MC202 pins used in QEII2C design * Future implementation

Note: SDA and SCK of I2C module, as well as QEA and QEB, must be assigned to 5-volt tolerant pins. For I2C, internal pull-up resistors to 5 Volts are activated by the wire library of Arduino, so no external resistors are required.

## A.2 Slave Software Development

The desired functionality of QEII2C shield is simple. When the master issues a read command, it captures the current position count and sends the value. When the master sends a write command followed by a 32-bit integer, that value is written to the counter register of QEI module.

Software on the QEII2C shield consists of the following routines

- **Main :** initialize the QEI and I2C module. Then enters the main loop, which does nothing but blinking the mode LED for the moment. Future implementation may include reading the mode switch status and take action.
- **I2C Slave :** an ISR that is invoked upon being addressed by the Arduino master. Reading and writing to position counter is embedded in this ISR.

Since the two important functions of the shield are QEI and I2C interfaces, we arrange our discussion accordingly.

### A.2.1 QEI Module Initialization

The PIC32EP256MC202 has only one QEI module, labeled QEI1. First we need to assign the QEA1 and QEB1 to RP pins of the microcontroller. On our prototype the pin RPI44 and RPI45 are chosen, so we write assignment code as follows

```
RPINR14bits.QEA1R = 44;  // QEA1 to RPI44

RPINR14bits.QEB1R = 45;  // QEB1 to RPI45
```

Next, we consider whether to use the digital filter in the module. This function is useful when high-frequency noise is present. To set up the filter with sampling rate equals system clock divided by 8, for example

```
QEI1IOCbits.QFDIV = 0b011;  // DIF 1:8 clock divide

QEI1IOCbits.FLTREN = 1;     // enable the DIF
```

The QEI module of PIC24EP has several other functions, but this is all we need for basic setup. Some are left to default values. The last step is to enable the module

```
QEI1CONbits.QEIEN = 1;  // enable the module
```

For easy management, it is recommended that all setup instructions be written to a function such as `initQEI()`

### A.2.2 Reading and Writing the Position Register

The special 32-bit register in QEI1 module used to keep position count is named `POS1CNT`. Since the bus structure of PIC24 is 16-bit, accessing the 32-bit register needs 2 consecutive reads/writes. With no data latch error could happen when, for example, the lower 16-bit is read first, but before the higher 16-bit is read, overflow happens that causes change in the higher 16-bit data. So after that data is read and combined with the lower 16-bit data to 32-bit position value, it is different from the actual value in the counter. This could happen with writing process as well.

To fix this problem, the QEI module in PIC24EP deploys two 16-bit special registers `POS1CNTL` and `POS1HLD` to transfer lower 16-bit (LSW: Least Significant Word) and higher 16-bit (MSW: Most Significant Word) of `POS1CNT`, respectively. The user program never reads or writes to/from `POS1CNT` directly.

Whenever a read to `POS1CNTL` is issued, the QEI module automatically writes the MSW of `POS1CNT` at that instant to `POS1HLD`. So the data in `POS1CNTL` and `POS1HLD` combined to a valid 32-bit position value.

Similarly, the writing process to `POS1CNT` starts from writing MSW to `POS1HLD` first. Later when the LSW is written to `POS1CNTL`, the value in `POS1HLD` is transferred to MSW of `POS1CNT` automatically. The data written to `POS1CNT` is valid.

On the software side, therefore, it is convenient to use a global variable of type `union` as follows

```
typedef union
{
    uint16_t half[2];
    uint32_t full;
} qeipvalue;
volatile qeipvalue QEIpVal;
```

This variable accommodates easy reading from `POS1CNT` of QEI1

```
long readENC(void)
{
    QEIpVal.half[0] = POS1CNTL; // read lsw
    QEIpVal.half[1] = POS1HLD;  // read msw from hold register
    if (SysFlag.ENCSense==CW) return QEIpVal.full;  // rotation
                              // sense dictated by SysFlag.ENCSense
    else return -QEIpVal.full;
}
```

as well as writing from `QEIpVal` to `POS1CNT`

```
void writeENC(long encvalue)
{
    if (SysFlag.ENCSense == CW) QEIpVal.full = encvalue;
    else QEIpVal.full = -encvalue;
    POS1HLD = QEIpVal.half[1]; // write msw to hold register
    POS1CNTL = QEIpVal.half[0];  // write lsw to POS1CNTL
}
```

Notice the `SysFlag.ENCSense` variable that allows the user to switch the encoder count direction by software This adds more flexibility to the shield since encoder structure varies.

### A.2.3 I2C Module Initialization

Unlike the A and B inputs of QEI module, the pins for I2C module cannot be remapped. Pin 14 and 15 are chosen because they are 5-V tolerant. So we have to use I2C2. First thing first, we must set configuration bit accordingly at the top of program

```
_FPOR(ALTI2C2_ON);       // enable alt i2c pin
```

Then configure pin 14 and 15 as open-drain outputs

```
ODCBbits.ODCB5=0;  // ASCL2
```

```
ODCBbits.ODCB6=0;  // ASDA2
```

For most of I2C functions used on the PIC24, we apply the code from [10]. Consult that book for some omitted detail such as variable type definition. Module configuration routine is written as follows

```
void configI2C2(uint16_t u16_FkHZ) {
  uint32_t u32_temp;
  u32_temp = (FCY/1000L)/((uint32_t) u16_FkHZ);
  u32_temp = u32_temp - FCY/10000000L - 1;
  I2C2BRG = u32_temp;
  I2C2CONbits.I2CEN = 1;
}
```

with FCY declared as 70 MHz. I2C2 speed is initialized to 100 KHz, compatible to the Arduino side

```
configI2C2(100);
```

The slave address is chosen as

```
#define QEII2CADDR 0x30      // I2C address of QEI module
```

then the module is initialized by this routine

```
void initI2CSlave(void) // initialize I2C as slave
{
    I2C2ADD = QEII2CADDR;  // initialize the address register
    _SI2C2IF = 0;
    _SI2C2IP = 1;
    _SI2C2IE = 1;
}
```

### A.2.4 State Machine for I2C Slave

It is explained quite clearly in [10] on I2C slave-master communication between 2 PIC24s. We adapt that concept to our case, with PIC24 as slave and Arduino as master. So the software structure in the slave remains quite the same with that described in [1]; i.e., the ISR for I2C2 consists of 5 following states

```
typedef enum {STATE_WAIT_FOR_ADDR, STATE_WAIT_FOR_WRITE_DATA,
        STATE_SEND_READ_DATA, STATE_SEND_READ_LAST } STATE;
```

Then the global state variable is initialized to

```
volatile STATE e_mystate = STATE_WAIT_FOR_ADDR;
```

The position counter data is 32 bits, while standard I2C protocol allows sending data byte by byte. So to avoid 4 shift operations, similar to the method explained earlier, we define union global variables `QEIData` and `QEICmd`

```
typedef union
{
    uint8_t b8[4];  // 1/4 of long variable
    long li32;     // keep full 32-bit position counter
} qeidata;
qeidata QEIData, QEICmd;
```

to keep current and command positions, respectively. To access the most significant byte of current position, for example, we read/write a byte to/from `QEIData.b8[3]`.

The core of QEII2C lies in the following I2C interrupt routine

```
void __attribute__((interrupt, auto_psv)) _SI2C2Interrupt(void)
{
    uint8_t u8_c;
    _SI2C2IF = 0;
    switch (e_mystate)   {
        case STATE_WAIT_FOR_ADDR:
            u8_c = I2C2RCV;  // clear RBF bit for address
            QEIData_index = 0;
            if (I2C2STATbits.R_W)   { // check R/W# bit of address
                                     // byte
                RDLED = ON;
                QEIData.li32=readENC();
                I2C2TRN = QEIData.b8[QEIData_index++]; //send first
                                                        //byte of data
                I2C2CONbits.SCLREL = 1;  // release clock line
                e_mystate = STATE_SEND_READ_DATA;
            }
            else  {
                WRLED = ON;
                e_mystate = STATE_WAIT_FOR_WRITE_DATA;
            }
            break;
```

```
        case STATE_WAIT_FOR_WRITE_DATA:
            // character arrived, place in buffer
            QEICmd.b8[QEIData_index++]=I2C2RCV;
            if (QEIData_index>3)   { // 4 bytes received
                writeENC(QEICmd.li32); // write to encoder
                WRLED = OFF;
                e_mystate = STATE_WAIT_FOR_ADDR;
            }
            break;
        case STATE_SEND_READ_DATA:
            // put data in transmit register
            I2C2TRN = QEIData.b8[QEIData_index++];
            I2C2CONbits.SCLREL = 1;   // release clock line
            if (QEIData_index>4)  // all 4 bytes sent
                e_mystate = STATE_SEND_READ_LAST;
            break;
        case STATE_SEND_READ_LAST: // last character finished TX
            RDLED = OFF;
            e_mystate = STATE_WAIT_FOR_ADDR;
            break;
        default:
            e_mystate = STATE_WAIT_FOR_ADDR;
    }
}
```

which is rather long, since function call from an interrupt service routine should be avoided as much as possible.

In this ISR a state machine for I2C slave communication is implemented. From the initial `STATE_WAIT_FOR_ADDR` state, if the master addresses this slave with a read command, the condition `if (I2C2STATbits.R_W)` is true. RDLED is turned on and the current encoder position is captured

```
RDLED = ON; // ON is defined as 1
QEIData.li32=readENC();
```

before the state jumps to `STATE_SEND_READ_DATA`, where the data in `QEIData` is sent to the master, starting from the most-significant byte.

```
I2C2TRN = QEIData.b8[QEIData_index++];
```

```
I2C2CONbits.SCLREL = 1;   // release clock line
```

So after being in this state for 4 consecutive interrupt cycles, the whole content in the 32-bit counter is sent. This condition is checked and the state jumps to STATE_SEND_READ_LAST

```
if (QEIData_index>4)  // all 4 bytes sent
        e_mystate = STATE_SEND_READ_LAST;
```

This last state does nothing other than turning the RDLED off and return to STATE_WAIT_FOR_ADDR.

On the other hand, when a write command is issued by the master, the condition if (I2C2STATbits.R_W) is false, so its else clause is executed. That turns on WRLED and causes state jump to STATE_WAIT_FOR_WRITE_DATA. There the 32-bit position command from the master is read byte by byte.

```
QEICmd.b8[QEIData_index++]=I2C2RCV;
```

After 4 consecutive cycles in this state QEICmd.li32 contains the whole position command. This condition is checked and  command position is written to the QEI counter register.

```
if (QEIData_index>3)   { // 4 bytes received
      writeENC(QEICmd.li32); // write to encoder
      WRLED = OFF;
      e_mystate = STATE_WAIT_FOR_ADDR;
}
```

WRLED is turned off, and the state returns to STATE_WAIT_FOR_ADDR.

### A.2.5 Configure PIC24EP to 70 MIPS Performance

The last software discussion on the PIC24EP slave is about how to tweak it to maximum achievable performance. The CPU does not run at 70 MIPS right out of the package. Some clock-switching instructions must be added during the initialization.

First, the configuration bit macro, tailored to internal FRC, is coded at the top of program.

```
_FOSCSEL(FNOSC_FRC);                // select internal FRC at POR
_FOSC(FCKSM_CSECMD & OSCIOFNC_ON & POSCMD_NONE);
                                    // enable clock switching
```

Then, somewhere in the main function before entering the infinite loop, write this code to setup the PLL and perform clock switching

```
// Configure PLL prescaler, PLL postscaler, PLL divisor (FRC mode)
PLLFBD = 74;   // M = 76
CLKDIVbits.PLLPOST = 0;      // N2 = 2
CLKDIVbits.PLLPRE=0;      // N1=2
// initiate clock switch to FRC oscillator with PLL (NOSC=0b01)
__builtin_write_OSCCONH(0x01);
__builtin_write_OSCCONL(0x01);
while (OSCCONbits.COSC != 0b001);
while (OSCCONbits.LOCK != 1);
```

See Microchip documents and [2] for detailed explanation.

## A.3 Arduino as I2C Master

When the QEII2C shield is mounted on the Arduino board, SCL and SDA pins of the AVR are connected to ASCL2 and ASDA2 of PIC24EP, respectively. Normally, both I2C lines must have pull-up resistors to VCC. This is done by internal pull-up resistors in the AVR when Wire library is used. Notice that the AVR on Arduino UNO R3 and PIC24EP operate at different voltage level (5 v.s. 3.3 volts). The I2C communication may not work properly if you use external pull-up to 3.3 volts. Internal pull-up to 5 volts by the AVR causes no problem to ASCL2 and ASDA2 of PIC24EP because the two pins are safe with 5V logic. (This is the reason to use I2C2 module. The ASCL1 pin of I2C1 is not 5V-tolerant!)

`qei_i2c.ino` in Listing A.1 demonstrates how to read/write encoder position data via I2C using the Wire library, which is included with the Arduino IDE. To focus only on the topics being discussed, the `setup()` and `loop()` structure of the sketch is maintained without using timer function. Only one additional function `cmdInt()` is implemented as command interpreter. The function accepts only single command `setenc`, which overwrites the position counter by a user-specified value. For example, after the command `setenc 1000`, the position counter in the QEI module should equal 1000.

```
// qei_i2c : by dew.ninja
// August 2016
// Test Quadrature Encoder Shield with i2c interface
// read position counter from module continuously
// display only when new received value is different from previous.
// Command:
//  - setenc d : set encoder position value to integer value d
```

```
#include <Wire.h>
String rcvdstring;   // string received from serial
String cmdstring;   // command part of received string
String  parmstring; // parameter part of received string
int parmvalint;        // parameter value
int newcmd = 0;     // flag when new command received
int sepIndex;       // index of seperator

typedef union
{
  byte b8[4];    // 1/4 of long variable
  long li32;
} qeidata;

qeidata QEIData, QEICmd;
long qeidata_old;
int qei_index=0;

void cmdInt(void);    // command interpreter function

void setup() {
  Wire.begin();  // join i2c bus (address optional for master)
  Serial.begin(9600);
}

void loop() {
  while (Serial.available() > 0 )   {
        rcvdstring = Serial.readString();
        newcmd = 1;
  }
  if (newcmd)   { // execute only when new command received
    cmdInt();   // invoke the interpreter
    newcmd = 0;
```

```
  }
  qeidata_old = QEIData.li32;    // save previous encoder count
  QEIData.li32 = 0;
  qei_index = 0;
  Wire.requestFrom(0x30, 5);  // request 5 bytes from slave device
                              // 0x30
  delay(10);
  while (Wire.available())  {  // slave may send less than requested
    QEIData.b8[qei_index++] = Wire.read();    // read a byte
  }
  if (qeidata_old != QEIData.li32)  // encoder value changed
    Serial.println(QEIData.li32);
  delay(100);    // delay 0.1 secc
}

// command interpreter implementation
void cmdInt(void)
{
    // find index of separator (blank character)
    sepIndex = rcvdstring.indexOf(' ');
    // extract command and parameter
    cmdstring = rcvdstring.substring(0, sepIndex);
    parmstring = rcvdstring.substring(sepIndex+1);
    parmvalint = parmstring.toInt();  // convert parameter to value
    // check if received command string is a valid command
    if (cmdstring.equalsIgnoreCase("setenc"))  {
      QEICmd.li32 = parmvalint;
      Serial.print("Encoder position counter set to ");
      Serial.println(QEICmd.li32);
      Wire.beginTransmission(0x30); // transmit to PIC24EP
      for (qei_index=0;qei_index<4;qei_index++) {//send 4 data bytes
        Wire.write(QEICmd.b8[qei_index]);
      }
      Wire.endTransmission();      // stop transmitting
      delay(10);
```

```
    }
    else
      Serial.println("Invalid command");
}
```

Listing A.1 `qei_i2c.ino` Arduino sketch example on how to read/write encoder position

A few remarks are on order.

- To use the Wire library, `Wire.h` must be included, and `Wire.begin()` is coded in `setup()`. No address is passed to `Wire.begin()` since Arduino acts as the master in this communication pair.
- The same union structure as in the other end is used to send each byte of the 4-byte long integer.
- Position counter is read in each loop execution. It is displayed on the serial monitor only when the value is changed.
- To read from I2C, first issue `Wire.requestFrom(address, n)` where n represents number of bytes. Then read each byte with `Wire.read()`. Note in Listing A.1 that 5 bytes are requested even only 4 are actually sent. (Choose n = 4 causes the loop to hang. I cannot explain this problem. It is safe to request more bytes than actual data because `Wire.available()` is checked in the next while loop.
- If the user issues a command, it is passed to `cmdInt()`. Encoder position register is replaced by the specified value if the command string matches `setenc`.

To test the sketch, create a simple setup as shown in Figure A.4 by attaching a handwheel to the shield. This is an input device commonly used in a CNC machine to move its axis to a specified location. The advantage is when we turn the dial one click, exactly one pulse for each A and B channel is generated. So, with the QEI module set up as x4, for each click, the counter value should increase or decrease by 4, depending on the turning direction.

Figure A.5 shows the result from running qei_i2c.ino. The output is captured from Serial Monitor window. The QEI position counter is initialized to 80000 at reset, to make sure that a 32-bit long integer is sent via I2C without error. After that we turn the handwheel in both directions to see the counter increases or decreases accordingly. Then we issue command setenc -1000 to see the counter changed to -1000. Turning the handwheel increases/decreases the negative value in the correct sense. Set the count once more to 0. Everything works as expected.

Bravo! We are satisfied with this QEII2C shield. It is ready to be used with motor control experiments in this book.

Figure A.4 testing the QEII2C shield with a handwheel

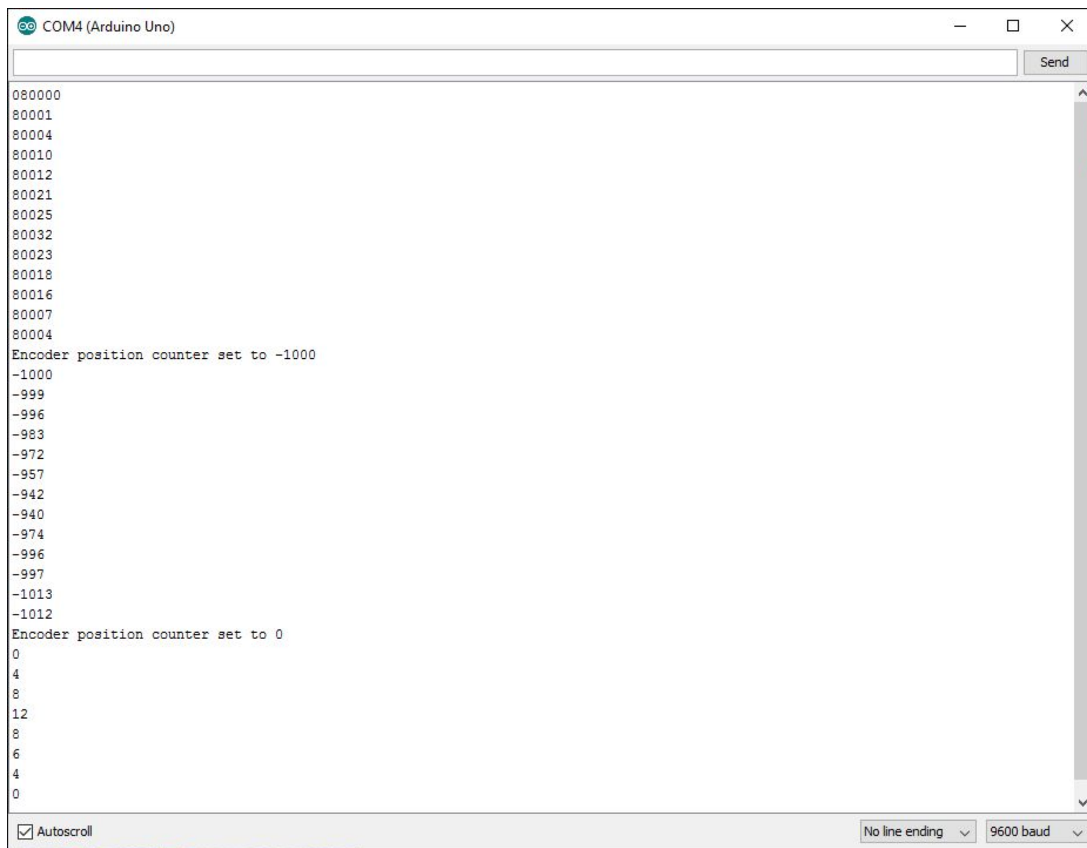


Figure A.5 Position readouts from the QEII2C shield