Serial Data Collection Team, SoftSys Final Report

Eric Schneider, Ankeet Mutha, Kaitlin Gallagher

May 8, 2014

1 Introduction

Our goal for this project was to learn to work with sensors using an Arduino. We wanted to read data off of one of a sensor. Because we were unable to make the sensors actually respond to our system, we made a simulated sensor in C that imitated the sensor's call and response system, providing the functionality we expected from the sensor. We set up an Arduino that can ping the simulated sensor with the required input, interpret the response, and output the simulated data.

2 Sensor

The sensor we were working with was a GE PQMII. It had an RS232 port that we could use to read and write serially to the sensor. The sensor responds to commands in the form of a 16 byte hex number and can perform a variety of actions, including returning incoming data. See figure 1 for example call and response.

Master Transmission	Bytes	Example	Description
Slave Address	1	11	message for slave 17
Function Code	1	08	loopback test
Diagnostic code	2	00 00	must be 00 00
Data	2	00 00	must be 00 00
CRC	2	E0 0B	CRC error code

Slave Response	Bytes	Example	Description
Slave Address	1	11	message from slave 17
Function Code	1	08	loopback test
Diagnostic Code	2	00 00	must be 00 00
Data	2	00 00	must be 00 00
CRC	2	E0 0B	CRC error code

Figure 1: Example of command and response for communication test. Table taken from GE digital energy product manual for PQMII.

3 Arduino Actions

For this project we were teamed up with Victoria and Greg. The breakdown of work was that our team would try to get data from the sensor and Victoria and Greg, the memory management team, would take that data, try to build it up in a data structure, and dump it to the computer when signaled. As such, we abstracted the Data Collection team's job like so:

```
Data *getData() {
    pingSensor()
    readData(buffer)
    word collectedData = decode(buffer)
    return dataConstructor(collectedData)
}
```

We'll go into more detail in the sections below, but at a basic level:

- **pingSensor** writes a hex value to the sensor over serial that is supposed to make the sensor write a value back
- **readData** scoops up all the data on the serial port
- **decode** takes the data from the sensor and extricates the data we care about from the larger string
- **dataConstructor** takes the two bytes of information that was decoded from the sensor output and inserts into the linked list structure desired by the memory management team

3.1 pingSensor - Serial Write

The sensor responds to commands with a very specific format: address, function code, data, CRC error code. Each field is a single byte, with the exception of the data field which varies depending on the function. Our code utilizes the loopback command, which acts as a communication test, and data collection, which requests data.

3.2 readData - Serial Read

Our readData section of the code is very simple. We create a byte array buffer and then loop through the elements of the buffer, reading one character of the serial port at a time into the buffer until there aren't any elements left. After decoding the data in the buffer we reset the buffer so that each element has a value of zero. As discussed in Section 2, the sensor has predefined structures for what sorts of data it will send back. The first byte will be the slave address, for example, the next will be the function address, etc. The readData buffer stores all of that information and passes it on to the decode function.

3.3 decode - Sort Data

The decoding section of the code takes the data in the buffer, parses the different sections, and returns the important parts as a 2 byte word. In the case of a loop-back test, this merely requires checking that the sensor returned the proper response. The data collection mode is slightly more complex. The first few bytes, which represent the address, function, and number of data bytes sent, are saved for error checking. Because we only ever request a singly data type at a time, the data section always consists of two bytes containing the upper and lower bits of a data value. These are combined and returned as a 2-byte word.

4 C Test Setup

When we plug the collector Arduino into the sensor the system is totally opaque to us, so we wanted to fully test the Arduino code beforehand and make sure that:

- The Arduino is sending out the correct hex keys specified in the sensor manual
- When the sensor sends back data, the Arduino can grab the correct information based on the expected return structure specified in the manual

We did this by making one of our computers into a fake sensor using C. The code (seen in Section 7.2) will open a serial port as a file directory, read the data coming in off the computer, and then write data back to the Arduino along the same serial port. We displayed the data from the Arduino as hex, allowing us to make sure the Arduino was sending the hex values that the manual said should be used. We also sent a hex string back to the Arduino either as a successful result for the ping test or as fake sensor data.

5 Results and Interface

Once we had our test rig working, we tried interfacing with the sensor. We were not able to get any response from the sensor despite various attempts and different changes we made. Instead we decided to use another computer as a "sensor" to generate results for Greg and Victoria in the format described above.

6 Conclusion and Further Work

We learned a lot from this project including further experience with Arduino and c. We also learned about some of the difficulties in working with real objects and how annoying they can be. If we were to take this project further, we would try harder to contact GE and figure out how to actually interface with their sensor. Additionally, we could try using different ports or different sensors to see where the problem lies.

7 Appendix: Code

7.1 Arduino Code

The Arduino code would send out a test ping (mode = 1) or a request for information (mode = 2). In mode 1 it would flash a green light when the test ping was returned correctly and in mode 2 it would display the extracted data. The getData function was made to be ported over to the memory management team's code.

```
#include <stdint.h>

typedef struct Data{
  word data;
} Data;

Data *getData(int mode);
void resetBuffer(byte *buf);
void pingSensor(int mode);
void readData(byte *buf);
word decode(byte *buf, int mode);
Data *dataConstructor(word data);

int green = 12;  // Port on the Arduino
int red = 11;    // Port on the Arduino
int white = 10;  // Port on the Arduino
const int len = 256;

void setup(){
  // start serial port at 9600 bps:
  Serial.begin(9600);
  pinMode(green, OUTPUT);
  pinMode(red, OUTPUT);
  pinMode(white, OUTPUT);
}

Data *getData(int mode){
  byte buf[len];
  resetBuffer(buf);
  pingSensor(mode);
  delay(250);
  readData(buf);
  word b = decode(buf, mode);
  return dataConstructor(b);
}

void resetBuffer(byte *buf) {
```

```
  // Sets all the data to something
  digitalWrite(white,LOW);
  digitalWrite(red,LOW);
  digitalWrite(green,LOW);
  for (int i=0;i<len;i++){
    buf[i] = 0;
  }
}
void pingSensor(int mode){
  int msgLen;
  uint8_t msg[16];

  if (mode == 2){ //3 phase real power
    msg[0] = 0x00; msg[1] = 0x03; msg[2] = 0x02; msg[3] = 0xF0;
    msg[4] = 0x00; msg[5] = 0x01; msg[6] = 0x9D; msg[7] = 0x8D;
    msgLen = 8;
  }
  else{ //loopback (default)
    msg[0] = 0x11; msg[1] = 0x08; msg[2] = 0x00; msg[3] = 0x00;
    msg[4] = 0x00; msg[5] = 0x00; msg[6] = 0xE0; msg[7] = 0x0B;
    msgLen = 8;
  }
  for (int i=0;i<msgLen;i++){
    Serial.write(msg[i]);
  }
}

void readData(byte *buf){
  int i=0;
  while(Serial.available() > 0) {
   buf[i] = Serial.read();
   i = i+1;
  }
}

word decode(byte *buf, int mode){
  word data = 65535;
  switch (mode){
  case 1: {  //loopback
    byte test[] = {1,1,1};
    if ((buf[0] == 0x00) && (buf[1] == 0x08) && (buf[6] == 0xE0)
                                              && (buf[7] == 0x0B)){
      digitalWrite(green,HIGH);
      return 0;
    }
```

```
    else{
     digitalWrite(red,LOW);
     return data;
    }
    break;
  }
  case 2: {  //read power
    byte addr = buf[0];
    byte func = buf[1];
    byte numdata = buf[2];
    byte dataHigh = buf[3];
    byte dataLow = buf[4];
    byte CRC = buf[5];
    //error checking
    if (0){
      digitalWrite(red,HIGH);
      return data;
    }
    digitalWrite(green,HIGH);
    data = word(dataHigh,dataLow);
    return data;
    break;
  }
  default:
    digitalWrite(white, HIGH);
    return data;
  }
}

Data *dataConstructor(word data){
  Data *newData = (Data *) malloc(sizeof(Data));
  newData->data = data;
  return newData;
}

void loop()
{
  Data *readData = getData(1);
  delay(1250);
  Serial.print("Read: ");
  Serial.println(readData->data);  // For debugging
  delay(1500);
}
```

7.2 C Fake Sensor Code

This code opens up a serial port in C and reads what is being written to that serial port, then writes values back. Though it could evaluate the request being sent from the Arduino and decide which values to send back based on that, we instead would simply comment and uncomment the values that the computer would send back when it read a string of the correct size from the Arduino.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <unistd.h>

#include <errno.h>
#include <termios.h>
#include <fcntl.h>

// Code largely taken from http://stackoverflow.com/questions/6947413
    // /how-to-open-read-and-write-from-serial-port-in-c

int
set_interface_attribs (int fd, int speed, int parity)
{
        struct termios tty;
        memset (&tty, 0, sizeof tty);
        if (tcgetattr (fd, &tty) != 0)
        {
                fprintf(stderr, "error %d from tcgetattr", errno);
                return -1;
        }

        cfsetospeed (&tty, speed);
        cfsetispeed (&tty, speed);

        tty.c_cflag = (tty.c_cflag & ~CSIZE) | CS8;     // 8-bit chars
        // disable IGNBRK for mismatched speed tests; otherwise receive break
        // as \000 chars
        tty.c_iflag &= ~IGNBRK;         // ignore break signal
        tty.c_lflag = 0;                // no signaling chars, no echo,
                                        // no canonical processing
        tty.c_oflag = 0;                // no remapping, no delays
        tty.c_cc[VMIN]  = 0;            // read doesn't block
        tty.c_cc[VTIME] = 5;            // 0.5 seconds read timeout

        tty.c_iflag &= ~(IXON | IXOFF | IXANY); // shut off xon/xoff ctrl
```

```
        tty.c_cflag |= (CLOCAL | CREAD);// ignore modem controls,
                                        // enable reading
        tty.c_cflag &= ~(PARENB | PARODD);      // shut off parity
        tty.c_cflag |= parity;
        tty.c_cflag &= ~CSTOPB;
        tty.c_cflag &= ~CRTSCTS;

        if (tcsetattr (fd, TCSANOW, &tty) != 0)
        {
                fprintf(stderr, "error %d from tcsetattr", errno);
                return -1;
        }
        return 0;
}

void
set_blocking (int fd, int should_block)
{
        struct termios tty;
        memset (&tty, 0, sizeof tty);
        if (tcgetattr (fd, &tty) != 0)
        {
                fprintf(stderr, "error %d from tggetattr", errno);
                return;
        }

        tty.c_cc[VMIN]  = should_block ? 1 : 0;
        tty.c_cc[VTIME] = 5;            // 0.5 seconds read timeout

        if (tcsetattr (fd, TCSANOW, &tty) != 0)
                fprintf(stderr, "error %d setting term attributes", errno);
}


char *portname = "/dev/ttyUSB0";

int
main() {
int fd = open (portname, O_RDWR | O_NOCTTY | O_SYNC);
if (fd < 0)
{
        fprintf(stderr, "error %d opening %s: %s", errno, portname, strerror (errno));
        return 0;
}
```

```
set_interface_attribs (fd, B9600, 0);  // set speed to 9600 bps, 8n1 (no parity)
set_blocking (fd, 0);                // set no blocking

int send_len = 8;
uint8_t send_data[] = {0x11, 0x08, 0x00, 0x00, 0x00, 0x00, 0xE0, 0x0B};

int buf_len = 256;
char buf[256];
int bytes_read;
int counter;
int i;

while (1) {
bytes_read = read (fd, buf, sizeof buf);    // read up to 100 characters if
                                            // ready to read
printf("bytes_read: %d \n", bytes_read);
                for (i=0; i<bytes_read; i++) {
                        //printf("Recieved: %d \n", (uint8_t) buf[i]);
                        //printf("Recieved: %c \n", (uint8_t) buf[i]);
                        printf("Recieved: %x \n", (uint8_t) buf[i]);
                }
printf("\n");

if (bytes_read >= 8) {
for (i=0; i<send_len; i++) {
printf("Sent: %x \n", (uint8_t) send_data[i]);
write (fd, &(send_data[i]), 1);
}
printf("\n");
}
usleep(500000);
}
}
```