ARL-TR-8259 ● Apr 2018

US Army Research Laboratory

Advanced Shutter Control for a Molecular Beam Epitaxy Reactor

by Ryan Enck and Eric Rong

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-8259 ● APR 2018

US Army Research Laboratory

Advanced Shutter Control for a Molecular Beam Epitaxy Reactor

by Ryan Enck
Sensors and Electron Devices Directorate, ARL

Eric Rong
The Johns Hopkins University, Baltimore, MD

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
April 2018	Technical Report	1 October 2016–30 September 2017

4. TITLE AND SUBTITLE
Advanced Shutter Control for a Molecular Beam Epitaxy Reactor

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Ryan Enck and Eric Rong

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
US Army Research Laboratory
ATTN: RDRL-SEE-I
2800 Powder Mill Road
Adelphi, MD 20783-1138

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-8259

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
An open-source hardware and software-based shutter controller solution was developed that communicates over Ethernet with our original equipment manufacturer (OEM) molecular beam epitaxy (MBE) reactor control software. An Arduino Mega microcontroller is the used for the brain of the shutter controller, while a custom-designed circuit board distributes 24-V power to each of the 16 shutter solenoids available on the MBE. Using Ethernet communications supported by the OEM software, the new shutter controller eliminates the need for proprietary interface cards inside the control computer.

15. SUBJECT TERMS
molecular beam epitaxy, MBE, shutter control, microcontroller, Ethernet

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Ryan Enck
Unclassified	Unclassified	Unclassified	UU	76	19b. TELEPHONE NUMBER (Include area code) (301) 394-1923

Contents

List of Figures

INTENTIONALLY LEFT BLANK.

1. Introduction

A molecular beam epitaxy (MBE) reactor is a system of stainless steel chambers, vacuum pumps, liquid nitrogen reservoirs, sample heaters, diagnostic equipment, and elemental sources used to grow semiconductor or metallic crystals. MBE reactors are used in government and private research laboratories, universities, and industry for the growth of crystals for basic physics and materials research through high-volume manufacturing. Most MBE reactors require an extensive set of electronics, including temperature-controlling power supplies, vacuum pump controllers, vacuum gauge controllers, flow sensors, an interlock system, a control computer, and much more.

1.1 MBE Reactor Design

An MBE reactor usually consists of 2 or 3 vacuum chambers connector in series, each with its own pumps, vacuum gauges, and specific purpose. The chamber vented most often is usually called the load lock. This chamber has pumps designed to evacuate the chamber from atmospheric pressure to high vacuum, or less than 1E-5 Torr. The next chamber in the sequence is often called the buffer chamber. This chamber is rarely vented to atmospheric pressure and therefore uses ion pumps designed to maintain an ultra-high-vacuum environment of less than 1E-8 Torr. Usually the highest vacuum (lowest pressure) chamber that vents the least often is called the growth chamber. This chamber contains the elemental sources and shutters, the sample heater, several different types of pumps, and liquid nitrogen reservoirs.

1.2 Shutters

MBE shutters are critical components of an MBE because they determine which elemental sources are used to grow a particular crystal. Without precisely timed shutter actuation, complex crystal structures could not be grown. Shutters usually consist of a tantalum metal or pyrolytic boron nitride ceramic blade that rapidly moves into and out of the atomic beam exiting each elemental source, either preventing or allowing the atomic flux to reach the substrate heater where the desired crystal is grown. Each shutter is attached to either a linear or rocking feedthrough. A rocking feedthrough uses a stem welded to a bellows that can pivot back and forth in one direction to move the shutter to the open and closed positions. A linear feedthrough simply is pushed in or pulled out to move the shutter between positions. Both shutter actuator types require a mechanism for converting electrical energy or pneumatic energy from a shutter controller into the physical motion that

moves the shutter. The existing MBE uses rocking shutters, whose movement is accomplished by using a solenoid valve to allow compressed air to enter one side or the other of a piston cylinder. Depending on which side of the cylinder receives air pressure, the piston moves toward the opposite side of the cylinder. This linear motion is converted in rotary motion by an axle with a rack and pinion that couples the axle to the piston movement. The axle then converts this rotary motion into linear motion of a shutter arm by using a screw that fixes the shutter arm to the axle near the outer edge. On the other side of the shutter arm, it is attached to the rocking feedthrough using a circular fastener containing a bushing, and the linear movement of the shutter arm pulls or pushes on the feedthrough arm, which is forced to pivot about an axis by a fulcrum on the feedthrough body.

2. Existing Shutter Control Technique

2.1 Hardware

The existing shutter control hardware for the MBE consists of a computer containing a National Instruments mixed-signal input/output (I/O) card (PCIe-6323), rack-mounted voltage conversion hardware, a system controller for interlocking the shutter commands, and a remote shutter box. When the control software requests a shutter to open, a software driver installed in the computer takes the control system command and raises the appropriate digital output line in the I/O card to 5 V. The voltage conversion hardware converts this 5-V signal to 24 V, which passes through the interlock system and, if permitted by the interlock conditions, causes the solenoid for the appropriate shutter to actuate, opening the shutter.

2.2 Software

The software necessary to control the shutter movement consists of the National Instruments NI-DAQ driver software and the proprietary MBE control software designed to interact with the driver and convert crystal growth recipes into a sequence of shutter movements.

2.3 Drawbacks

The existing shutter control system has limitations in operability, portability, and future upgradability of the computer. The control software that converts recipes for growth into precisely timed shutter sequences also is intended to confirm that with each shutter actuation request the shutter position sensors detect that movement. However, the existing shutter control system reroutes the signal requesting a shutter

movement back into the input for that shutter that is supposed to report on the shutter movement. In other words, the existing shutter controller tricks the control software into thinking that it confirmed a real shutter movement. Since no true mechanism exists for detecting shutter movement in hardware, shutters can freeze to the LN2 paneling or solenoids can become stuck, interfering with the intended growth of the crystal without any record of the event occurring.

The portability of the current shutter control system is impacted by the use of the mixed-signal I/O cards. These cards require motherboard slots, and therefore the computer must be a desktop PC. Also, if motherboard slots change as technology advances, eventually motherboards with the necessary slots will be difficult or impossible to obtain. This also means that the control system cannot be installed on a laptop or in a compact desktop configuration that lacks the space and appropriate motherboard slots. Also, since the National Instruments card requires drivers to communicate with the control software, there is a risk that the drivers will not be compatible with future operating systems, potentially requiring the replacement of the entire control system computer and control software, currently costing approximately $20,000. If the company providing the control software is no longer in business, an entirely new control system hardware and software would have to be purchased, which could cost more than $100,000.

3. New Shutter Control System

The new shutter control system addresses the concerns with operability, portability, and future upgradability of the existing shutter control system. The new shutter controller has 2 inputs for each shutter for sensing the proximity of the shutter actuator to 2 position sensors. The sensors are located at the shutter movement extremes so that 3 states can be detected: opened, closed, and intermediate. Once implemented, this new capability will prevent unknown shutter malfunctions from going undetected.

While the existing control software has drivers for communicating with legacy hardware that requires the use of National Instruments cards, it also has the ability to communicate over Ethernet using a protocol called Modbus Transmission Control Protocol (TCP). By configuring the control software to talk to an Ethernet-based shutter controller and then monitoring the communications over that port, the details of the protocol were determined. The new shutter control system was designed to be compatible with the native instrument driver for the control software. This approach ensures that the existing, and very likely future, control software will be able to control the new shutter control system. By switching the communication protocol to Modbus TCP instead of proprietary third-party drivers created by

National Instruments, any future computer with an Ethernet port can communicate with the new shutter controller. A laptop or compact desktop lacking Peripheral Component Interconnect slots can also be used since no motherboard slots are required.

The new shutter control system uses an Arduino microcontroller for the software logic and libraries. The Arduino development platform is open source and well supported by a large community of software developers and hobbyists. The open-source approach ensures that future users of the shutter controller can make modifications and improvements leveraging the work of the larger community.

3.1 Hardware Design

3.1.1 Requirements

- 16 outputs at 24 V (300 mA per channel) for shutter solenoid control
- Thirty-two 24-V inputs for shutter position sensing
- Thirty-two 5-V outputs for driving 16 bicolor (red, green) LEDs for displaying shutter position
- Modbus TCP Ethernet connectivity
- Automatic recovery from infinite loops and software crashes

3.1.2 Microprocessor

The specific Arduino hardware used in the new shutter controller is the Arduino Mega. This Arduino variant uses the ATmega1280 microcontroller, an 8-bit AVR Reduced Instruction Set Computer (RISC)-based microcontroller that contains 128-kB In-System Programming (ISP) flash memory, 8 kB of Static Random-Access Memory (SRAM), and 4 kB of Electrically Erasable Programmable Read-Only Memory (EEPROM). This larger processor has 86 general-purpose I/O lines, several counters/timers, 16-channel Pulse-Width Modulation (PWM), 16-channel 10-bit analog/digital (A/D) converter, 4 Universal Asynchronous Receiver-Transmitters (UARTs), and an inter-integrated circuit (I^2C) bus and 5 Serial Peripheral Interface (SPI) buses. Despite the low cost ($50), the device achieves a throughput of 16 million instructions per second at 16 MHz and is compatible with 5-V logic.

3.1.3 Port Expansion

Although the Arduino Mega uses the Atmega1280 microprocessor, some of the digital I/O ports are consumed by other onboard hardware. As a result, the Arduino Mega only has 54 usable digital I/O lines. Since this is not enough ports to handle all of the shutter controller I/O requirements, a 32-channel port expander from Abel Electronics that uses 2 Microchip MCP23017s was chosen. This port expander breakout board communicates over the I^2C bus, is compatible with 5-V logic, and can be configured as inputs or outputs with or without pull-up resistors.

3.1.4 Voltage Conversion

3.1.4.1 Shutter Output

Since the Arduino Mega can only output 5-V signals at 40 mA, a voltage and current conversion technique is used to drive the shutter solenoids. The new shutter controller uses optically isolated solid-state relays (SSRs) for voltage conversion and increased current capacity. The output voltage conversion for each shutter port, from 5 to 24 V, is performed by a G3VM-41AY SSR from Omron. Each SSR can switch 2 A of current at 40 V. Across each shutter solenoid output a fly-back diode is included to suppress voltage spikes that form when the solenoid, which is an inductor, is de-energized.

3.1.4.2 Shutter Position Sensing

Shutter position sensing requires sensing a 24-V input from reed switches attached to the extreme ends of the shutter actuator piston cylinder. When the shutter actuator has moved to one end of travel, a magnetic element on the piston causes a reed switch contactor to close. This causes an internal p-n-p transistor to turn on, which pulls the collector of the transistor from ground to 24 V. Sensing the 24-V signal is accomplished using an LAA710 dual optically isolated SSR. Each relay has a current limiting resistor of 1.2 kohms, which limits the input current of the relay to 20 mA. This current was chosen based on the minimum on-current of 10 mA for this SSR and the maximum recommended input current of 50 mA. When the SSR is turned on by the reed switch transistor, the output side of the SSR pulls a pin on the port expansion breakout board to 5 V. This input change can be detected by the Arduino using the I^2C bus.

3.1.5 Modbus TCP Communication over Ethernet

The Arduino Mega does not have Ethernet communication functionality built-in; however, there are “shields” that enable the end user to enhance the core capabilities of the Arduino hardware. The new shutter controller is using the Wiznet

W5500 Ethernet Shield. Since this shield uses the W5500 Ethernet chipset, it is compatible with the Ethernet2 Arduino library. By using this shield and the Arduino library, Ethernet connectivity becomes trivial.

3.1.6 Watchdog Timer

The watchdog timer is a countdown timer with its own internal hardware clock that, once configured, operates independently of the Arduino sketch code. Once the watchdog timer reaches zero, it initiates an immediate hardware reset. The purpose of the watchdog timer is to make sure that if the Arduino sketch ever hangs due to an infinite loop error, or if a memory leak occurs that causes the microprocessor to crash, the watchdog timer will time out and reset the device. This reset does not occur during normal operation because each time the main loop executes, the watchdog timer count is reset to the starting count, preventing the reset from occurring.

3.1.7 Schematics

Several schematic diagrams have been included to document the electrical design of the principal hardware function blocks.

3.1.7.1 Shutter Output Block

The shutter output block (Fig. 1) is repeated 16 times throughout the circuit board for each shutter output. As described, the shutter output block converts the Arduino pin voltage and current to a voltage and current sufficient to drive the shutter solenoid. In addition, one override switch is placed in series with the Arduino output so that the user can always disable an output or place the output in the open position permanently. This user interlocking and override functionality was desired in case the new shutter control system malfunctioned or became damaged and the shutters needed to be controlled manually.

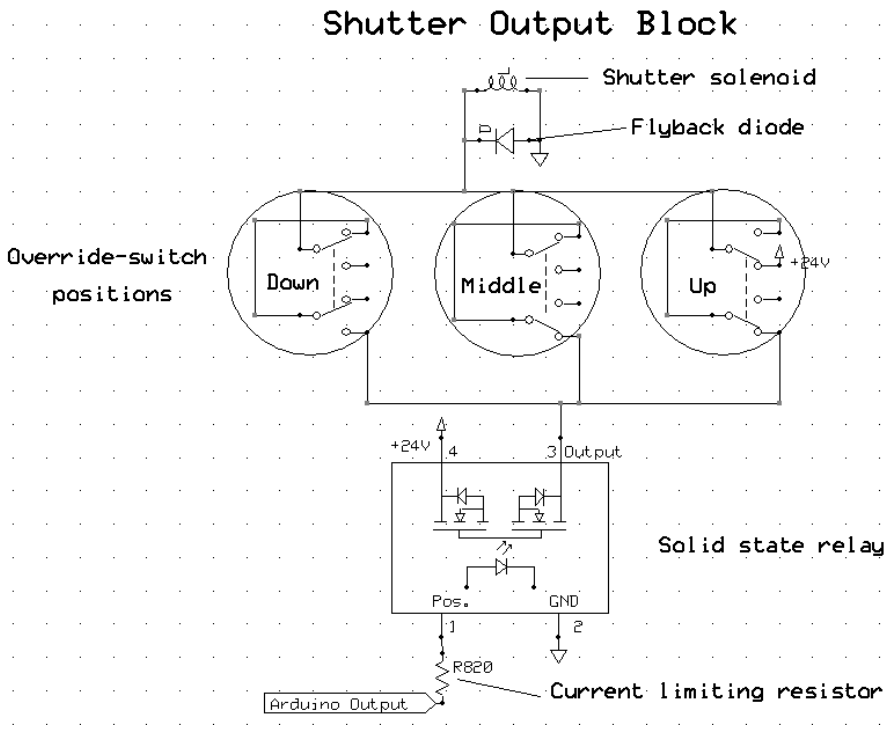

Fig. 1 Schematic of the shutter output block

3.1.7.2 Shutter Position Sensing Block

The shutter position input sensing block (Fig. 2) is repeated 32 times through the circuit board. As mentioned, this hardware block converts the 24-V signals from the shutter sensor transistors to 5-V signals that can be sensed by the port expander hardware.

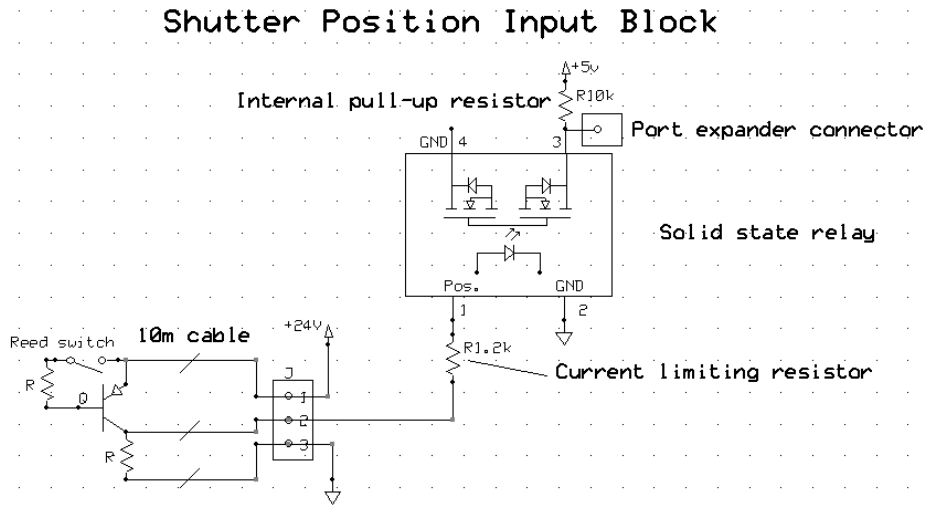

Fig. 2 Schematic of the shutter position input block

3.1.7.3 Bicolor Red/Green LED Block

The bicolor red/green LED block (Fig. 3) simply enables the Arduino to safely drive a bicolor LED whose diodes are wired in parallel with opposite polarity. A current limiting resistor ensures that each diode receives approximately 10 mA of current.

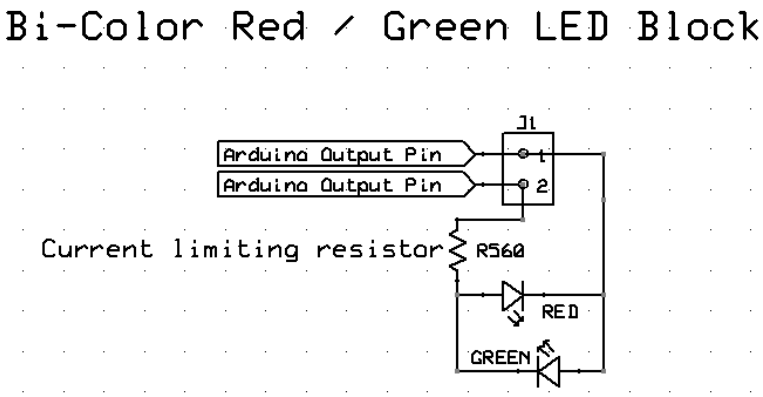

Fig. 3 Schematic of the bicolor red/green block

3.1.8 Circuit Board

The circuit board was designed using a free program from Express PCB. The circuit board contains 4 layers (top trace, power plane, ground plane, and bottom trace) and was sized to conform to the ProtoPro board requirements. The ProtoPro board format places limits on board area and length in exchange for very economical and quick circuit-board manufacturing. Illustrations of the top trace (Fig. 4), bottom trace (Fig. 5), screen printing (Fig. 6), and entire board layout (Fig. 7) follow.

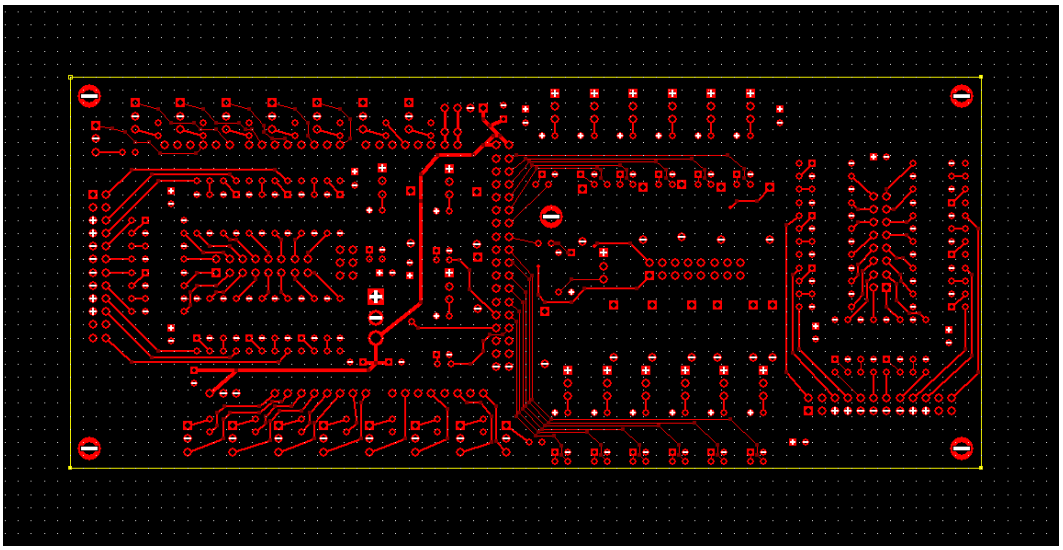

Fig. 4 Top trace

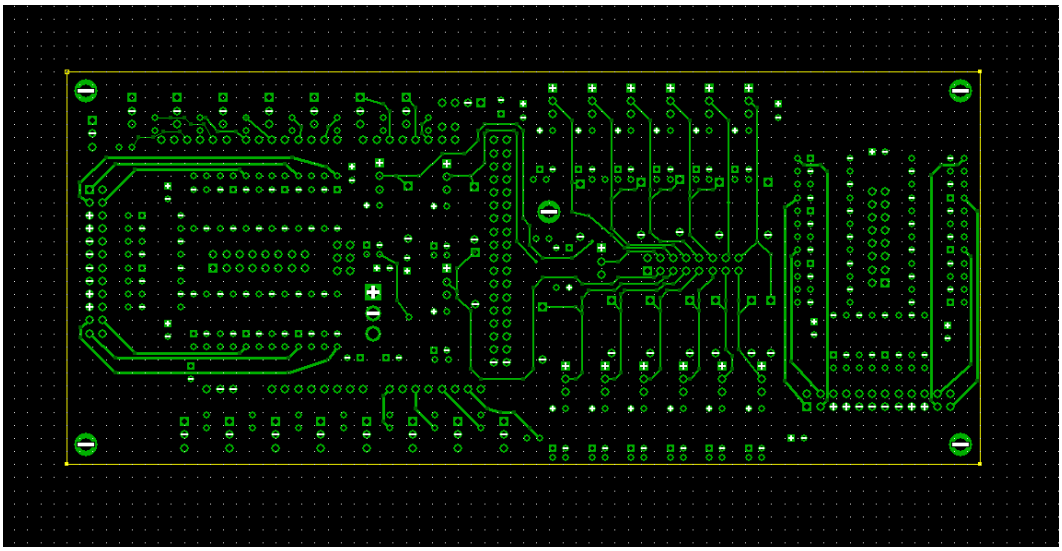

Fig. 5 Bottom trace

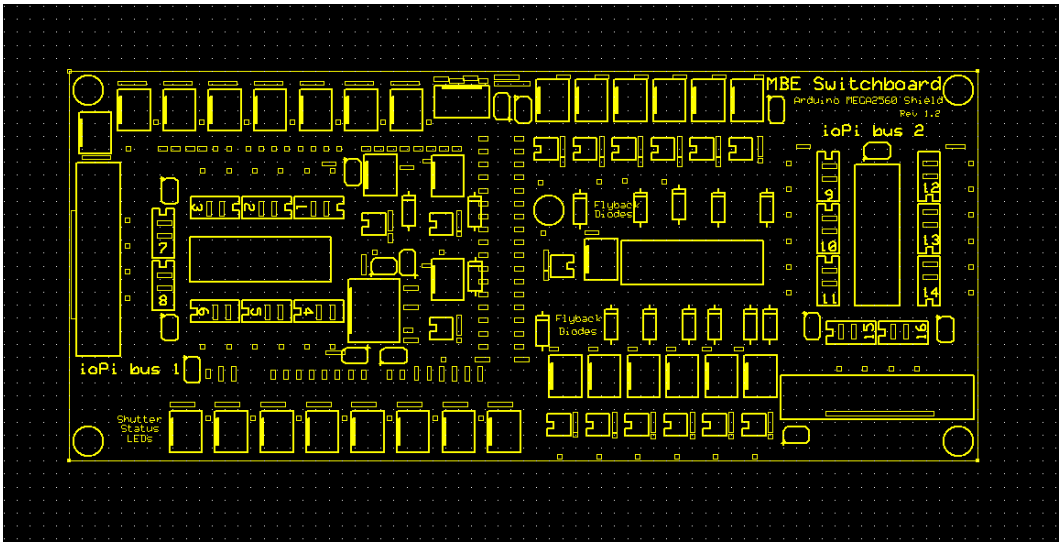

Fig. 6 Screen printing

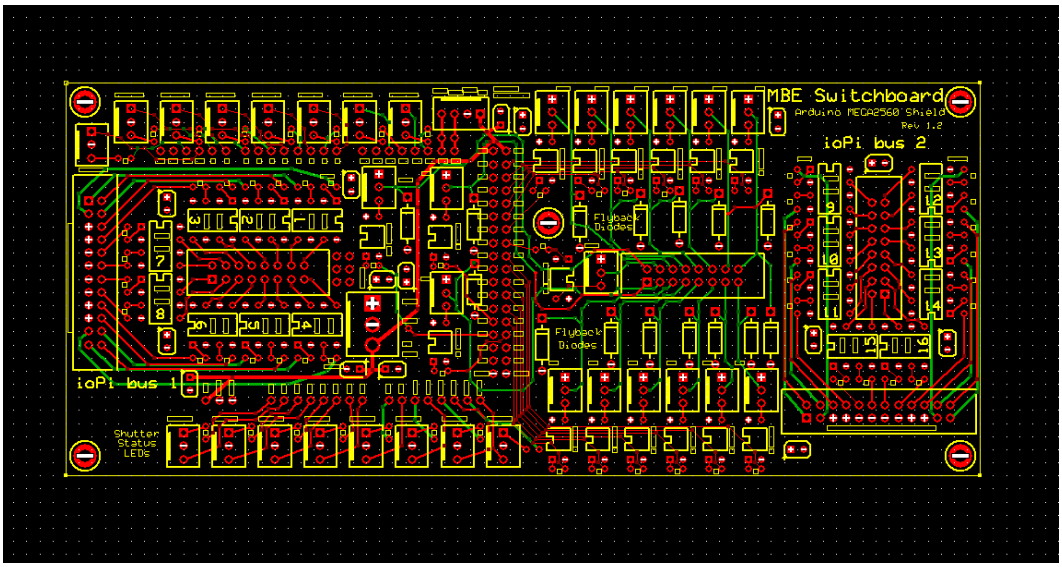

Fig. 7 Entire board layout

3.1.9 Enclosure

The enclosure (front view, Fig. 8; rear view, Fig. 9) was chosen based on the following criteria:

- Sufficient space for the following:
  - Assembled circuit board
  - Port expander breakout board
  - All cabling
  - 16 switches
  - 16 LEDs
  - USB connector
  - Ethernet connector
  - Shutter output connector
  - Shutter position input connector
  - Power supply connector
- Sufficiently small to fit on a computer desk
- Portable enough to carry around the MBE lab as needed

Fig. 8 Front view

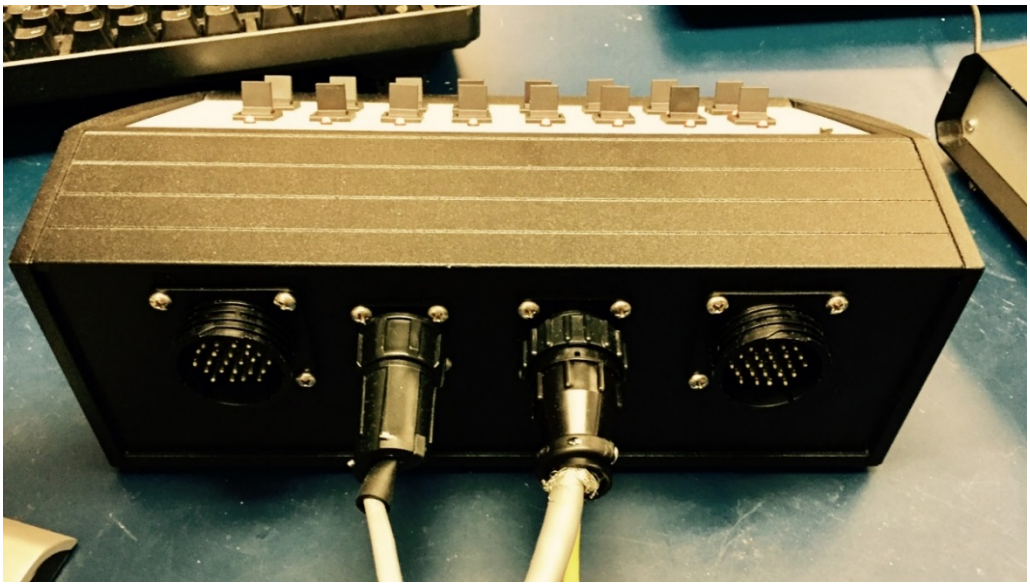

Fig. 9 Rear view

3.2 Software Design

As described previously, the software design platform was the Arduino integrated development environment (IDE). This IDE was largely chosen due to the large number of freely available software libraries, large array of hardware expansion shields, and well supported forums. The software language supported by the IDE is C++, a commonly used programming language with a very large community knowledgebase.

3.2.1 Arduino Sketch

The Arduino sketch contains the main program, library dependencies, variable declarations, object instantiation, the Setup() function, and the main program loop. Each component of the sketch is discussed in detail in the following.

3.2.2 Libraries

The libraries (see the Appendix) used to support the new shutter controller are as follows:

- SPI: The SPI library is necessary for the Arduino to communicate with the Ethernet shield over the SPI bus. This library handles the timing and configuration according to the SPI protocol.
- Ethernet2: The Ethernet2 library is also needed for the Ethernet shield but includes the W5500 Ethernet chipset specific hardware definitions and functions.
- Mudbus (modified): The modified Mudbus library is needed to receive and process communication packets, conforming to the Modbus TCP communication protocol. The library functions handle packet dissection, function code execution, packet construction and response, and error handling.
- Wire: The Wire library is needed for communication with the port expander shield using the I^2C bus.
- ABElectronics_IOPi: The ABElectronics_IOPi library is used to configure the MCP23017 port expander chips. This library includes functions for setting pin direction, reading and writing to I/O pins, and configuring interrupts.
- Avr/wdt: The Avr/wdt library is needed to access and configure the watchdog timer registers on the Atmega1280 microcontroller.
- Flasher_modified: The Flasher_modified library is an object-oriented library used to simplify the process of controlling the LEDs. Each LED object can be referenced by an object array and can be controlled simply by calling the appropriate method. The original library was modified to allow instantiating arrays of flasher objects and to modify the library to work with bicolor LEDs.

3.2.3 Global Variables

- const int greenLED[16] array: This array of 16 integers stores the pin number of the green LED cathode (red LED anode) for the corresponding LED number. Array element 0 corresponds to LED 1, which displays the status of shutter 1.
- const int redLED[16] array: This array of 16 integers stores the pin number of the red LED cathode (green LED anode) for the corresponding LED number. Array element 0 corresponds to LED 1, which displays the status of shutter 1.
- const int shutterControl[16] array: This array of 16 integers stores the pin number of the shutter output pin for the corresponding shutter number. Array element 0 corresponds to shutter 1. This variable is initialized to 1111111111111111 (65535 in decimal).
- Unsigned int shutterOpenStatus is a 16-bit unsigned integer, where each individual bit stores whether the open shutter reed switch detects the shutter actuator magnetic element. This variable is initialized to 1111111111111111 (65535 in decimal).
- Unsigned int shutterCloseStatus is a 16-bit unsigned integer, where each individual bit stores whether the open shutter reed switch detects the shutter actuator magnetic element.
- Uint8_t mac[6] is an array of 6 bytes, which identifies the unique hardware identity of the shutter controller.
- Uint8_t ip[4] is an array of 4 bytes, which sets the IP address of the shutter controller.
- Uint8_t gateway[4] is an array of 4 bytes, which sets the gateway address of the shutter controller.
- Uint8_t subnet[4] is an array of 4 bytes, which sets the subnet mask of the shutter controller.

3.2.4 Object Instantiation

- Mudbus Mb creates an instance of the Mudbus class and executes the constructor.
- ABElectronics_IOPi iopi1(0x20) creates an instance of the ABElectronics_IOPi class and passes the I^2C bus address (0x20) to the new instance.

- ABElectronics_IOPi iopi2(0x21) creates an instance of the ABElectronics_IOPi class and passes the I^2C bus address (0x21) to the new instance.

3.2.5 Custom Functions

- watchdogSetup() temporarily disables all interrupts, configures and initializes the watchdog timer, and then re-enables the interrupts.
- changeShutter() accepts the shutter ID (0, 1, 2, etc.) and the requested status (open = 1, close = 0) and performs a digital write to the appropriate pin of the Arduino Mega.
- Float2Bytes() accepts a 32-bit single precision floating point number and the address of an array of four bytes. The function segments the 32-bit float into 4 bytes and stores those bytes in the byte array whose address was passed in during the function call.

3.2.6 Setup()

The Setup() function runs one time at program load or reboot.

Wire.begin() initializes the Wire library to enable use of the I^2C bus.

Serial.begin() initializes the serial port library to allowing debugging over the serial port monitor built into the Arduino IDE.

Ethernet.begin(mac, ip, gateway, subnet) initializes the Ethernet2 library, which attempts to register the device (server) with the router or host computer.

The following for loop creates an instance of the Flasher class for each of 16 LEDs that display the status of a corresponding shutter:

```
For ( int I = 0; I < 16; i++ ) {
        LED [ I ] = Flasher();
}
```

The following for loop assigns the appropriate pins and blink times for each LED object:

```
For ( int I = 0; I < 16; i++) {
        LED[ I ] setPins ( redLED[i], greenLED[i] ) ;
        LED[ I ] setBlink ( LEDONTIME, LEDOFFTIME ) ;
}
```

The following line of code stores the number 1 in a special Modbus register, which is monitored by the control software and used to indicate whether the shutter controller is operational:

```
Mb.R [ DIGITALSHIM ] = 1;
```

The following for loop sets the output mode for each shutter control pin in the Arduino Mega to OUTPUT:

```
For ( int I = 0; I < 16; i++) {
        pinMode( shutterControl [ I ], OUTPUT );
}
```

Since the port expander shield has 2 MCP23017 integrated circuits (ICs) and each IC has 2 ports, the following 8 lines of code set each port as an input and turns on the internal pullup resistors for each pin on each port:

```
Iopi1.SetPortDirection( 0, 0xFF );
Iopi1.SetPortDirection( 1, 0xFF );
Iopi2.SetPortDirection( 0, 0xFF );
Iopi2.SetPortDirection( 1, 0xFF );
Iopi1. SetPortPullups ( 0, 0xFF );
Iopi1. SetPortPullups ( 1, 0xFF );
Iopi2. SetPortPullups ( 0, 0xFF );
Iopi2. SetPortPullups ( 1, 0xFF );
```

The following 2 lines of code initialize the Modbus registers to 65535. These registers are read by the control software to indicate the status of each shutter position sensor.

```
Mb.R [ shutterStatusInputOffset ] = 65535;
Mb.R [ shutterStatusInputOffset + 1 ] = 65535;
```

The following If statement and internal for loops runs a routine at startup that causes each shutter status LED to illuminate red and then green, which allows the user to visually determine if any LEDs have failed:

```
If (RUN_LED_TEST) {
        For ( int I = 0; I < 16; i++ ) {
                LED[i].Red();
                Delay(50);
        }

        For ( int I = 0; I < 16; i++ ) {
                LED [ I ].Green();
```

```
            Delay ( 50 );
        }
        delay( 1000 );
}
```

The last step in the Setup() function is to initialize the watchdog timer:

```
watchdogSetup();
```

3.2.7 Main Loop()

The main loop runs endlessly; with each loop completion, variables that were created inside the main loop are destroyed and the memory they consumed is returned to the microprocessor for future use.

The first task for the main loop is to reset the watchdog timer to prevent the Arduino from rebooting:

```
Wdt_reset();
```

The next 5 lines of code create local variables that are used throughout the main loop:

```
Boolean shutterOpened[16];
Boolean shutterClosed[16];
Boolean shutterCommand[16];
unsigned int shutterOpenedStatusRegister = 0;
unsigned int shutterClosedStatusRegister = 0;
```

The next line of code calls the Mudbus object Run method. This method checks for packets form the Ethernet input buffer and processes the data.

```
Mb.Run();
```

The next If statement checks to see if the shutter controller is using the shutter position feedback hardware. If true, the ReadPort methods are called for each ABElectronics_IOPi object and the returned values are concatenated into two 16-bit words containing the reed switch status for each shutter.

```
If ( !FAKE_SHUTTER_RESPONSE ) {
          byte iopi1port0 = iopi1.ReadPort(0);
        byte iopi1port1 = iopi1.ReadPort(1);
          shutterOpenStatus = iopi1Port1 * 256 | iopi1Port0
        byte iopi2port0 = iopi2.ReadPort(0);
```

```
        byte iopi2port1 = iopi2.ReadPort(1);
        shutterCloseStatus = iopi2Port1 * 256 | iopi2Port0
}
```

The following For loop reads the Modbus register values from the location in memory where the control software stores shutter commands. It also loads the shutter open and close status words into arrays to make the code more readable in subsequent sections.

```
For ( int I = 0; I < 8; i++) {
        shutterCommand[ I ] = bitRead ( Mb.R [shutterOutputOffset ], I );
         shutterCommand[ I + 8 ] = bitRead ( Mb.R [ shutterOutputOffset ], I + 8
);
         shutterOpened[ I ] = bitRead ( shutterOpenStatus, 2 * I – 2 );
         shutterOpened[ I + 8 ] = bitRead (shutterOpenStatus, 2 * I – 2 );
         shutterClosed[ I ] = bitRead (shutterCloseStatus, 2 * I – 2 );
         shutterClosed[ I + 8 ] = bitRead (shutterCloseStatus, 2 * I – 2 );
}
```

The following For loop acts on the received shutter commands from the control software and sets the LED condition (red, green, flashing red, flashing green, or flashing red and green) based on the shutter position sensors:

```
For ( int I = 0; I < 16; i++) {
         changeShutter ( I, shutterCommand [ I ] );
         if ( FAKE_SHUTTER_RESPONSE ) {
                  shutterOpened [ I ] = shutterCommand [ I ];
                  shutterClosed [ I ] = !shutterCommand [ I ];
         }
         If ( shutterCommand [ I ] == 1 ) {
                  If ( ( shutterOpened [ I ] ) && ( shutterClosed [ I ] ) ) {
                           LED [ I ].Green();
                  } else {
                           If ( ( !shutterOpened [ I ] ) && ( shutterClosed [ I ] ) ) {
                                    LED [ I ].FlashGreen();
                           } else {
                                    LED [ I ].FlashRedGreen();
                           }
                  }
         } else if (shutterCommand [ I ] == 0 ) {
                  If ( ( !shutterOpened [ I ] ) && ( shutterClosed [ I ] ) ) {
                           LED [ I ].Red();
                  } else {
                           If ( ( shutterOpened [ I ] ) && ( !shutterClosed [ I ] ) ) {
                                    LED [ I ].FlashRed();
                           } else {
```

```
                                    LED [ I ].FlashRedGreen();
                                }
                            }
                        }
}
```

The final For loop reconstructs the shutter open and closed status arrays back into 16-bit words that can be loaded into the Modbus registers that store the shutter status:

```
For ( int I = 0; I < 16; i++) {
        shutterOpenedStatusRegister = (shutterOpenedStatusRegister << 1 ) |
shutterOpened [ I ];
        shutterClosedStatusRegister = (shutterClosedStatusRegister << 1 ) |
shutterClosed [ I ];
 }
```

The final 2 lines of code store the shutter status words in the Modbus registers for retrieval by the control software:

```
Mb.R [ shutterStatusInputOffset ] = shutterOpenedStatusRegister;
Mb.R [shutterStatusInputOffset + 1 ] = shutterOpenedStatusRegister;
```

4. Conclusion

By leveraging open-source hardware and software and the large and well-supported Arduino community, we have developed a shutter control system that is customizable, more portable, and more compatible with future computer hardware. We have eliminated the need for proprietary components, allowing any computer with an Ethernet port or Ethernet adapter the ability to control the MBE shutters. Future efforts will expand upon this concept to provide even more capabilities and flexibility.

Appendix. Libraries

A-1 Libraries

A-1.1 ABElectronics_IOPi.h

```
/*
  ABElectronics_IOPi.h - Library for IOPi board.
  Created by Brian Dorey, January 7th 2016.
  Released into the public domain.
*/

#ifndef ABElectronics_IOPi_h
#define ABElectronics_IOPi_h

#include "Arduino.h"


class ABElectronics_IOPi
{
        public:
                // constructor
                ABElectronics_IOPi(byte address);
                // functions
                void Connect();
                void SetPinDirection(byte pin, bool direction);
                void SetPortDirection(byte port, byte direction);
                void SetPinPullup(byte pin, bool value);
                void SetPortPullups(byte port, byte value);
                void WritePin(byte pin, bool value);
                void WritePort(byte port, byte value);
                bool ReadPin(byte pin);
                byte ReadPort(byte port);
                void InvertPort(byte port, byte polarity);
                void InvertPin(byte pin, bool polarity);
                void MirrorInterrupts(byte value);
                void SetInterruptPolarity(byte value);
                void SetInterruptType(byte port, byte value);
                void SetInterruptDefaults(byte port, byte value);
                void SetInterruptOnPort(byte port, byte value);
                void SetInterruptOnPin(byte pin, bool value);
                byte ReadInterruptStatus(byte port);
                byte ReadInterruptCapture(byte port);
                void ResetInterrupts();
        private:
                bool isConnected;
                // Define registers values from datasheet
                byte IODIRA = 0x00;  // IO direction A - 1= input 0 = output
```

```
        byte IODIRB = 0x01;  // IO direction B - 1= input 0 = output
        byte IPOLA = 0x02; // Input polarity A - If a bit is set, the
corresponding GPIO register bit will reflect the inverted value on the pin.
        byte IPOLB = 0x03; // Input polarity B - If a bit is set, the
corresponding GPIO register bit will reflect the inverted value on the pin.
        byte GPINTENA = 0x04; // The GPINTEN register controls the
interrupt-on-change feature for each pin on port A.
        byte GPINTENB = 0x05; // The GPINTEN register controls the
interrupt-on-change feature for each pin on port B.
        byte DEFVALA = 0x06; // Default value for port A - These bits set
the compare value for pins configured for interrupt-on-change. If the associated
pin level is the opposite from the register bit, an interrupt occurs.
        byte DEFVALB = 0x07; // Default value for port B - These bits set
the compare value for pins configured for interrupt-on-change. If the associated
pin level is the opposite from the register bit, an interrupt occurs.
        byte INTCONA = 0x08; // Interrupt control register for port A.  If
1 interrupt is fired when the pin matches the default value, if 0 the interrupt is
fired on state change
        byte INTCONB = 0x09; // Interrupt control register for port B.  If
1 interrupt is fired when the pin matches the default value, if 0 the interrupt is
fired on state change
        byte IOCON = 0x0A;  // see datasheet for configuration register
        byte GPPUA = 0x0C;  // pull-up resistors for port A
        byte GPPUB = 0x0D;  // pull-up resistors for port B
        byte INTFA = 0x0E; // The INTFA register reflects the interrupt
condition on the port A pins of any pin that is enabled for interrupts. A set bit
indicates that the associated pin caused the interrupt.
        byte INTFB = 0x0F; // The INTFB register reflects the interrupt
condition on the port B pins of any pin that is enabled for interrupts. A set bit
indicates that the associated pin caused the interrupt.
        byte INTCAPA = 0x10; // The INTCAP register captures the
GPIO port A value at the time the interrupt occurred.
        byte INTCAPB = 0x11; // The INTCAP register captures the GPIO
port B value at the time the interrupt occurred.
        byte GPIOA = 0x12; // data port A
        byte GPIOB = 0x13;  // data port B
        byte OLATA = 0x14; // output latches A
        byte OLATB = 0x15; // output latches B

        // variables
        byte port_a_dir = 0x00;  // port a direction
        byte port_b_dir = 0x00;  // port b direction
        byte portaval = 0x00;  // port a value
        byte portbval = 0x00;  // port b value
        byte porta_pullup = 0x00; // port a pull-up resistors
        byte portb_pullup = 0x00; // port a pull-up resistors
```

```
                byte porta_polarity = 0x00;  // input polarity for port a
                byte portb_polarity = 0x00; // input polarity for port b
                byte intA = 0x00; // interrupt control for port a
                byte intB = 0x00; // interrupt control for port a

      byte config = 0x22; // initial configuration - see IOCON page in the
MCP23017 datasheet for more information.
                byte address = 0x20;
                char updateByte(char byte, char bit, char value);
                byte ReadI2CByte(byte value1);
                void WriteI2CByte(byte value1, byte value);
                bool CheckIntBit(int value, byte position);
                bool CheckBit(byte value, byte position);
                char UpdateByte(char byte, char bit, char value);
};
#endif
```

A-1.2 ABElectronics_IOPi.cpp

```
#include <Arduino.h>

#include <Wire.h>

#include "ABElectronics_IOPi.h"


ABElectronics_IOPi::ABElectronics_IOPi(byte icaddress)

{


  address = icaddress;




}


void ABElectronics_IOPi::SetPinDirection(byte pin, bool direction) {

        pin = (byte)(pin - 1);


        if (pin >= 0 && pin < 8)

        {
```

```
            port_a_dir = UpdateByte(port_a_dir, pin, direction);
            WriteI2CByte(IODIRA, port_a_dir);
        }
        else if (pin >= 8 && pin < 16)
        {
            port_b_dir = UpdateByte(port_b_dir, (byte)(pin - 8), direction);
            WriteI2CByte(IODIRB, port_b_dir);
        }
        else
        {
            // catch all for invalid pin
        }
    }
/// <summary>
    /// Set the direction for an IO port.  You can control the direction of all 8 pins on a port by sending a single byte value.
    /// Each bit in the byte represents one pin so for example 0x0A would set pins 2 and 4 to inputs and all other pins to outputs.
    /// </summary>
    /// <param name="direction">Direction for all pins on the port.  1 = input, 0 = output</param>
    /// <param name="port">0 = pins 1 to 8, 1 = pins 9 to 16</param>
    void ABElectronics_IOPi::SetPortDirection(byte port, byte direction)
    {
        switch (port)
        {
            case 0:
                WriteI2CByte(IODIRA, direction);
                port_a_dir = direction;
                break;
            case 1:
```

```
            WriteI2CByte(IODIRB, direction);
            port_b_dir = direction;
            break;
        default:
            // default
            break;
    }

}
/// <summary>
/// Set the internal 100K pull-up resistors for an individual pin
/// </summary>
/// <param name="pin">1 to 16</param>
/// <param name="value">true = enabled, false = disabled</param>
void ABElectronics_IOPi::SetPinPullup(byte pin, bool value)
{
    pin = (byte)(pin - 1);

    if (pin >= 0 && pin < 8)
    {
        porta_pullup = UpdateByte(porta_pullup, pin, value);
        WriteI2CByte(GPPUA, porta_pullup);
    }
    else if (pin >= 8 && pin < 16)
    {
        portb_pullup = UpdateByte(portb_pullup, (byte)(pin - 8), value);
        WriteI2CByte(GPPUB, portb_pullup);
    }
    else
```

```
        {
            // default

        }
    }
    /// <summary>
    /// set the internal 100K pull-up resistors for the selected IO port
    /// </summary>
    /// <param name="port">0 = pins 1 to 8, 1 = pins 9 to 16</param>
    /// <param name="value">number between 0 and 255 or 0x00 and
0xFF</param>
    void ABElectronics_IOPi::SetPortPullups(byte port, byte value)
    {
        switch (port)
        {
            case 0:
                porta_pullup = value;
                WriteI2CByte(GPPUA, value);
                break;
            case 1:
                portb_pullup = value;
                WriteI2CByte(GPPUB, value);
                break;
            default:
                 // default
                 break;
        }
    }
    /// <summary>
    /// write to an individual pin
```

```
        /// </summary>
        /// <param name="pin">1 - 16</param>
        /// <param name="value">0 = logic low, 1 = logic high</param>
        void ABElectronics_IOPi::WritePin(byte pin, bool value)
        {
            pin = (byte)(pin - 1);
            if (pin >= 0 && pin < 8)
            {
                portaval = UpdateByte(portaval, pin, value);
                WriteI2CByte(GPIOA, portaval);
            }
            else if (pin >= 8 && pin < 16)
            {
                portbval = UpdateByte(portbval, (byte)(pin - 8), value);
                WriteI2CByte(GPIOB, portbval);
            }
            else
            {
                // default
            }
        }
        /// <summary>
        /// write to all pins on the selected port.
        /// </summary>
        /// <param name="port">0 = pins 1 to 8, 1 = pins 9 to 16</param>
        /// <param name="value">number between 0 and 255 or 0x00 and
0xFF</param>
        void ABElectronics_IOPi::WritePort(byte port, byte value)
        {
            switch (port)
```

```
    {
        case 0:
            WriteI2CByte(GPIOA, value);
            portaval = value;
            break;
        case 1:
            WriteI2CByte(GPIOB, value);
            portbval = value;
            break;
        default:
             // default
             break;
    }
}
/// <summary>
/// read the value of an individual pin.
/// </summary>
/// <param name="pin">1 - 16</param>
/// <returns>0 = logic level low, 1 = logic level high</returns>
bool ABElectronics_IOPi::ReadPin(byte pin)
{
    pin = (byte)(pin - 1);
    if (pin >= 0 && pin < 8)
    {
        portaval = ReadI2CByte(GPIOA);
        return CheckBit(portaval, pin);
    }
    else if (pin >= 8 && pin < 16)
    {
```

```
        portbval = ReadI2CByte(GPIOB);
        return CheckBit(portbval, (byte)(pin - 8));
    }
    else
    {
        // default
    }
}
/// <summary>
/// read all pins on the selected port.
/// </summary>
/// <param name="port">0 = pins 1 to 8, 1 = pins 9 to 16</param>
/// <returns>returns number between 0 and 255 or 0x00 and 0xFF</returns>
byte ABElectronics_IOPi::ReadPort(byte port)
{
    switch (port)
    {
        case 0:
            portaval = ReadI2CByte(GPIOA);
            return portaval;
        case 1:
            portbval = ReadI2CByte(GPIOB);
            return portbval;
        default:
            // default
            break;
    }
}
/// <summary>
```

```
/// invert the polarity of the pins on a selected port.
/// </summary>
/// <param name="port">0 = pins 1 to 8, 1 = pins 9 to 16</param>
/// <param name="polarity">0x00 - 0xFF (0 = same logic state of the input pin, 1 = inverted logic state of the input pin)</param>

void ABElectronics_IOPi::InvertPort(byte port, byte polarity)
{
   switch (port)
   {
      case 0:
         WriteI2CByte(IPOLA, polarity);
         porta_polarity = polarity;
         break;
      case 1:
         WriteI2CByte(IPOLB, polarity);
         portb_polarity = polarity;
         break;
      default:
          // default
          break;
   }
}
/// <summary>
/// invert the polarity of the selected pin.
/// </summary>
/// <param name="pin">1 to 16</param>
/// <param name="polarity">False = same logic state of the input pin, True = inverted logic state of the input pin</param>
void ABElectronics_IOPi::InvertPin(byte pin, bool polarity)
```

```
{
    pin = (byte)(pin - 1);
    if (pin >= 0 && pin < 8)
    {
        porta_polarity = UpdateByte(portaval, pin, polarity);
        WriteI2CByte(IPOLA, porta_polarity);
    }
    else if (pin >= 8 && pin < 16)
    {
        portb_polarity = UpdateByte(portbval, (byte)(pin - 8), polarity);
        WriteI2CByte(IPOLB, portb_polarity);
    }
    else
    {
        // default
    }
}
/// <summary>
/// Sets the mirror status of the interrupt pins.
/// </summary>
/// <param name="value">0 = The INT pins are not mirrored. INTA is associated with PortA and INTB is associated with PortB. 1 = The INT pins are internally connected</param>
void ABElectronics_IOPi::MirrorInterrupts(byte value)
{
    switch (value)
    {
        case 0:
            config = UpdateByte(config, 6, false);
            WriteI2CByte(IOCON, config);
```

```
            break;
        case 1:
            config = UpdateByte(config, 6, true);
            WriteI2CByte(IOCON, config);
            break;
        default:
            // default
            break;
    }
}
/// <summary>
/// This sets the polarity of the INT output pins
/// </summary>
/// <param name="value">1 = Active - high. 0 = Active - low.</param>
void ABElectronics_IOPi::SetInterruptPolarity(byte value)
{
    switch (value)
    {
        case 0:
            config = UpdateByte(config, 1, false);
            WriteI2CByte(IOCON, config);
            break;
        case 1:
            config = UpdateByte(config, 1, true);
            WriteI2CByte(IOCON, config);
            break;
        default:
            // default
            break;
```

```
    }
}
/// <summary>
/// Sets the type of interrupt for each pin on the selected port. 1 = interrupt is fired when the pin matches the default value. 0 = the interrupt is fired on state change
/// </summary>
/// <param name="port">0 = pins 1 to 8, 1 = pins 9 to 16</param>
/// <param name="value">number between 0 and 255 or 0x00 and 0xFF</param>
void ABElectronics_IOPi::SetInterruptType(byte port, byte value)
{
    switch (port)
    {
        case 0:
            WriteI2CByte(INTCONA, value);
            break;
        case 1:
            WriteI2CByte(INTCONB, value);
            break;
        default:
             // default
             break;
    }
}
/// <summary>
/// These bits set the compare value for pins configured for interrupt-on-change on the selected port. If the associated pin level is the opposite from the register bit, an interrupt occurs.
/// </summary>
/// <param name="port">0 = pins 1 to 8, 1 = pins 9 to 16</param>
```

```
        /// <param name="value">number between 0 and 255 or 0x00 and
0xFF</param>
        void ABElectronics_IOPi::SetInterruptDefaults(byte port, byte value)
        {
            switch (port)
            {
                case 0:
                    WriteI2CByte(DEFVALA, value);
                    break;
                case 1:
                    WriteI2CByte(DEFVALB, value);
                    break;
                default:
                     // default
                     break;
            }
        }
        /// <summary>
        /// Enable interrupts for the pins on the selected port.
        /// </summary>
        /// <param name="port">0 = pins 1 to 8, 1 = pins 9 to 16</param>
        /// <param name="value">number between 0 and 255 or 0x00 and
0xFF</param>
        void ABElectronics_IOPi::SetInterruptOnPort(byte port, byte value)
        {
            switch (port)
            {
                case 0:
                    WriteI2CByte(GPINTENA, value);
                    intA = value;
```

```
            break;
        case 1:
            WriteI2CByte(GPINTENB, value);
            intB = value;
            break;
        default:
            // default
            break;
    }
}
/// <summary>
/// Enable interrupts for the selected pin.
/// </summary>
/// <param name="pin">1 to 16</param>
/// <param name="value">0 = interrupt disabled, 1 = interrupt enabled</param>
void ABElectronics_IOPi::SetInterruptOnPin(byte pin, bool value)
{

    pin = (byte)(pin - 1);
    if (pin >= 0 && pin < 8)
    {
        intA = UpdateByte(intA, pin, value);
        WriteI2CByte(GPINTENA, intA);
    }
    else if (pin >= 8 && pin < 16)
    {
        intB = UpdateByte(intB, (byte)(pin - 8), value);
        WriteI2CByte(GPINTENB, intB);
    }
```

```
    else
    {
        // default

    }
}
/// <summary>
/// read the interrupt status for the pins on the selected port.
/// </summary>
/// <param name="port">0 = pins 1 to 8, 1 = pins 9 to 16</param>
byte ABElectronics_IOPi::ReadInterruptStatus(byte port)
{
    switch (port)
    {
        case 0:
            return ReadI2CByte(INTFA);
        case 1:
            return ReadI2CByte(INTFB);
        default:
            // default
            break;
    }
}
/// <summary>
/// read the value from the selected port at the time of the last interrupt trigger.
/// </summary>
/// <param name="port">0 = pins 1 to 8, 1 = pins 9 to 16</param>
byte ABElectronics_IOPi::ReadInterruptCapture(byte port)
{
```

```
    switch (port)
    {
        case 0:
            return ReadI2CByte(INTCAPA);
        case 1:
            return ReadI2CByte(INTCAPB);
        default:
            // default
            break;
    }
}
/// <summary>
/// Reset the interrupts A and B to 0
/// </summary>
void ABElectronics_IOPi::ResetInterrupts()
{
    ReadInterruptCapture(0);
    ReadInterruptCapture(1);
}


        /// <summary>
/// Updates the value of a single bit within a byte and returns the updated byte
/// </summary>
/// <param name="value">The byte to update</param>
/// <param name="position">Position of the bit to change</param>
/// <param name="bitstate">The new bit value</param>
/// <returns></returns>


    char ABElectronics_IOPi::UpdateByte(char byte, char bit, char value)
```

```
    { /* internal method for setting the value of a single bit within a byte */
            if (value == 0) {
                    return (byte &= ~(1 << bit));
            } else {
            return (byte |= 1 << bit);
            }

    }

/// <summary>
/// Checks the value of a single bit within a byte
/// </summary>
/// <param name="value">The value to query</param>
/// <param name="position">The bit position within the byte</param>
/// <returns></returns>
bool ABElectronics_IOPi::CheckBit(byte value, byte position)
{
    // internal method for reading the value of a single bit within a byte
    return (value & (1 << position)) != 0;
}

bool ABElectronics_IOPi::CheckIntBit(int value, byte position)
{
    // internal method for reading the value of a single bit within a byte
    return (value & (1 << position)) != 0;
}
```

```
/// <summary>
/// Writes a single byte to an I2C device.
/// </summary>
/// <param name="bus">I2C device</param>
/// <param name="register">Address register</param>
/// <param name="value">Value to write to the register</param>
void ABElectronics_IOPi::WriteI2CByte(byte value1, byte value)
{
   Wire.beginTransmission(address);
                        Wire.write(value1);
                        Wire.write(value);
                        Wire.endTransmission();

}

/// <summary>
/// Read a single byte from an I2C device
/// </summary>
/// <param name="bus">I2C device</param>
/// <param name="register">Address register to read from</param>
/// <returns></returns>
byte ABElectronics_IOPi::ReadI2CByte(byte value1)
{
                        Wire.beginTransmission(address);
                        Wire.write(value1);
                        Wire.endTransmission();

                        byte val = 0x00;
                        Wire.requestFrom((int)address, 1);
```

```
                while (Wire.available() <= 0) {


                }
                  val = Wire.read();
                Wire.endTransmission();


        return val;
    }
```

A-1.3 Mudbus_modified.h

```
/*
   Mudbus_modified.h - an Arduino library for a Modbus TCP slave.
   Copyright (C) 2011  Dee Wykoff

   This program is free software: you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation, either version 3 of the License, or
   (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See
the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

//#define MbDebug
```

```
#include "Arduino.h"

#include <SPI.h>
#include <Ethernet2.h>

#ifndef Mudbus_h
#define Mudbus_h

#define MB_N_C_0x 100 //Max coils for Modbus is 100 due to limited memory
#define MB_N_I_1x 100 //Max inputs for Modbus is 100 due to limited memory
#define MB_N_IR_3x 64 //Max 16 bit input registers is 64 due to limited memory
#define MB_N_HR_4x 600 //Max 16 bit holding registers is 64 due to limited memory
#define MB_PORT 502

class Mudbus
{
public:
   Mudbus();
   void Run();
   bool C[MB_N_C_0x];
   bool I[MB_N_I_1x];
   int  IR[MB_N_IR_3x];
   int  R[MB_N_HR_4x];
   bool Active, JustReceivedOne;
   unsigned long PreviousActivityTime;
   int Runs, Reads, Writes, TotalMessageLength, MessageStart, NoOfBytesToSend;
   uint8_t Exception;
```

```
private:
   uint8_t ByteReceiveArray[160];
   uint8_t ByteSendArray[160];
   uint8_t SaveArray[160];
   int FC;
   void SetFC(int fc);
   void PopulateSendBuffer(uint8_t *SendBuffer, int NoOfBytes);
   void buffer_restore();
   void buffer_save();
};

#endif

/* Speculations on Modbus message structure:
*********************************************
**********Master(PC) request frames***********
00 ID high            0
01 ID low             1
02 Protocol high      0
03 Protocol low       0
04 Message length high  0
05 Message length low   6 (6 bytes after this)
06 Slave number       1
07 Function code
08 Start address high   maybe 0
09 Start address low    maybe 0
10 Length high          maybe 125 or Data high if write
11 Length low           maybe 125 or Data low if write
*********************************************
```

**********Slave(Arduino) response frames******

00 ID high echo / 0

01 ID low echo / slave ID 1

02 Protocol high echo

03 Protocol low echo

04 Message length high echo

05 Message length low num bytes after this

06 Slave number echo

07 Function code echo or 80h+echo for exception

08 Start address high num bytes of data

09 Data high

10 Data low

****

*/

A-1.4 Mudbus.cpp

/*

Mudbus.cpp - an Arduino library for a Modbus TCP slave.

Copyright (C) 2011 Dee Wykoff

This program is free software: you can redistribute it and/or modify

it under the terms of the GNU General Public License as published by

the Free Software Foundation, either version 3 of the License, or

(at your option) any later version.

This program is distributed in the hope that it will be useful,

but WITHOUT ANY WARRANTY; without even the implied warranty of

MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

GNU General Public License for more details.

```
   You should have received a copy of the GNU General Public License
   along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/
//#define MbDebug
//#define MbRunsDebug
#include "Mudbus_modified.h"

EthernetServer MbServer(MB_PORT);

Mudbus::Mudbus()
{
}

void Mudbus::Run()
{
   Runs = 1 + Runs * (Runs < 999);

   //******************** Read from socket ****************
   EthernetClient client = MbServer.available();
   if(client.available())
   {
      Reads = 1 + Reads * (Reads < 999);
      int i = 0;
      while(client.available())
      {
         ByteReceiveArray[i] = client.read();
         i++;
```

```
        }
        TotalMessageLength = i;
        NoOfBytesToSend = 0;
        MessageStart = 0;
#ifdef MbDebug
        for (i=0; i<TotalMessageLength; i++) {
           Serial.print(ByteReceiveArray[i],HEX);
           Serial.print(" ");
        }
        Serial.println(" received");
        Serial.print("MessageLength = ");
        Serial.println(TotalMessageLength);
#endif
        SetFC(ByteReceiveArray[7]);  //Byte 7 of request is FC
        JustReceivedOne = true;
        if(!Active)
        {
           Active = true;
           PreviousActivityTime = millis();
#ifdef MbDebug
           Serial.println("Mb active");
#endif
        }
     }
     if(millis() > (PreviousActivityTime + 60000))
     {
        if(Active)
        {
           Active = false;
```

```
#ifdef MbDebug
          Serial.println("Mb not active");
#endif
       }
   }


   int Start, WordDataLength, ByteDataLength, CoilDataLength, MessageLength;


   while (FC != 0) {
      //**1***************** Read Coils **********************
      if(FC == 1)
      {
         Start = word(ByteReceiveArray[8 + MessageStart],ByteReceiveArray[9 +
MessageStart]);
         CoilDataLength = word(ByteReceiveArray[10 +
MessageStart],ByteReceiveArray[11 + MessageStart]);
         if((Start+CoilDataLength) > MB_N_C_0x){
            FC += 128;
            Exception = 2;
            break;
         }
         ByteDataLength = CoilDataLength / 8;
         if(ByteDataLength * 8 < CoilDataLength) ByteDataLength++;
         CoilDataLength = ByteDataLength * 8;
#ifdef MbDebug
         Serial.print(" MB_FC_READ_COILS_0x S=");
         Serial.print(Start);
         Serial.print(" L=");
         Serial.println(CoilDataLength);
#endif
```

```
        buffer_save();
        ByteReceiveArray[5 + MessageStart] = ByteDataLength + 3; //Number of
bytes after this one.
        ByteReceiveArray[8 + MessageStart] = ByteDataLength;     //Number of
bytes after this one (or number of bytes of data).
        for(int i = 0; i < ByteDataLength ; i++)
        {
           for(int j = 0; j < 8; j++)
           {
              bitWrite(ByteReceiveArray[9 + i + MessageStart], j, C[Start + i * 8 +
j]);
           }
        }
        MessageLength = ByteDataLength + 9;
        PopulateSendBuffer(&ByteReceiveArray[MessageStart],
MessageLength);
        Writes = 1 + Writes * (Writes < 999);
        FC = 0;
        buffer_restore();
     }

     //**2********** Read descrete Inputs **********
     else if(FC == 2) //Arduino does not seem to like handling enum as int.
     {
        Start = word(ByteReceiveArray[8 + MessageStart],ByteReceiveArray[9 +
MessageStart]);
        CoilDataLength = word(ByteReceiveArray[10 +
MessageStart],ByteReceiveArray[11 + MessageStart]);
        if((Start+CoilDataLength) > MB_N_I_1x){
           FC += 128;
           Exception = 2;
           break;
```

```
        }
        ByteDataLength = CoilDataLength / 8;
        if(ByteDataLength * 8 < CoilDataLength) ByteDataLength++;
        CoilDataLength = ByteDataLength * 8;
#ifdef MbDebug
        Serial.print(" MB_FC_READ_INPUTS_1x S=");
        Serial.print(Start);
        Serial.print(" L=");
        Serial.println(CoilDataLength);
#endif
        buffer_save();
        ByteReceiveArray[5 + MessageStart] = ByteDataLength + 3; //Number of
bytes after this one.
        ByteReceiveArray[8 + MessageStart] = ByteDataLength;     //Number of
bytes after this one (or number of bytes of data).
        for(int i = 0; i < ByteDataLength ; i++)
        {
           for(int j = 0; j < 8; j++)
           {
              bitWrite(ByteReceiveArray[9 + i + MessageStart], j, I[Start + i * 8 +
j]);
           }
        }
        MessageLength = ByteDataLength + 9;
        PopulateSendBuffer(&ByteReceiveArray[MessageStart],
MessageLength);
        Writes = 1 + Writes * (Writes < 999);
        FC = 0;
        buffer_restore();
      }
```

```
//**3**************** Read Holding Registers ******************
else if(FC == 3)
{
   Start = word(ByteReceiveArray[8 + MessageStart],ByteReceiveArray[9 +
MessageStart]);
                  if (Start == 256) {
                        Start = 0;
                        } else if (Start == 768) {
                              Start = 100;
                              } else if (Start == 12288) {
                                    Start = 200;
                                    } else if (Start == 12289) {
                                          Start = 202;
                                          } else if (Start == 12290) {
                                                Start = 204;
                                                } else if (Start ==
12291) {
                                                      Start = 206;
                                                      } else if (Start
== 12292) {
                                                            Start =
208;
                                                            } else
if (Start == 12293) {
      Start = 210;
      } else if (Start == 12294) {
      Start = 212;
      } else if (Start == 12295) {
```

```
                Start = 214;
                } else if (Start == 12296) {
                    Start = 216;
                    } else if (Start == 12297) {
                        Start = 218;
                        } else if (Start == 12416) {
                            Start = 220;
                            }
        WordDataLength = word(ByteReceiveArray[10 +
MessageStart],ByteReceiveArray[11 + MessageStart]);
        if((Start+WordDataLength) > MB_N_HR_4x){
          FC += 128;
          Exception = 2;
          break;
        }
        ByteDataLength = WordDataLength * 2;
#ifdef MbDebug
        Serial.print(" MB_FC_READ_REGISTERS_4x S=");
        Serial.print(Start);
        Serial.print(" L=");
        Serial.println(WordDataLength);
#endif
        buffer_save();
        ByteReceiveArray[5 + MessageStart] = ByteDataLength + 3; //Number of
bytes after this one.
```

```
        ByteReceiveArray[8 + MessageStart] = ByteDataLength;     //Number of
bytes after this one (or number of bytes of data).
        for(int i = 0; i < WordDataLength; i++)
        {
          ByteReceiveArray[ 9 + i * 2 + MessageStart] = highByte(R[Start + i]);
          ByteReceiveArray[10 + i * 2 + MessageStart] =  lowByte(R[Start + i]);
        }
        MessageLength = ByteDataLength + 9;
        PopulateSendBuffer(&ByteReceiveArray[MessageStart],
MessageLength);
        Writes = 1 + Writes * (Writes < 999);
        FC = 0;
        buffer_restore();
      }

      //**4***************** Read Input Registers ******************
      else if(FC == 4)
      {
        Start = word(ByteReceiveArray[8 + MessageStart],ByteReceiveArray[9 +
MessageStart]);
        WordDataLength = word(ByteReceiveArray[10 +
MessageStart],ByteReceiveArray[11 + MessageStart]);
        if((Start+WordDataLength) > MB_N_IR_3x){
          FC += 128;
          Exception = 2;
          break;
        }
        ByteDataLength = WordDataLength * 2;
#ifdef MbDebug
        Serial.print(" MB_FC_READ_INPUT_REGISTERS_3x S=");
        Serial.print(Start);
```

```
        Serial.print(" L=");
        Serial.println(WordDataLength);
#endif
        buffer_save();
        ByteReceiveArray[5 + MessageStart] = ByteDataLength + 3; //Number of bytes after this one.
        ByteReceiveArray[8 + MessageStart] = ByteDataLength;     //Number of bytes after this one (or number of bytes of data).
        for(int i = 0; i < WordDataLength; i++)
        {
          ByteReceiveArray[ 9 + i * 2 + MessageStart] = highByte(IR[Start + i]);
          ByteReceiveArray[10 + i * 2 + MessageStart] =  lowByte(IR[Start + i]);
        }
        MessageLength = ByteDataLength + 9;
        PopulateSendBuffer(&ByteReceiveArray[MessageStart], MessageLength);
        Writes = 1 + Writes * (Writes < 999);
        FC = 0;
        buffer_restore();
      }


    //**5**************** Write Coil **********************
    else if(FC == 5)
    {
        Start = word(ByteReceiveArray[8 + MessageStart],ByteReceiveArray[9 + MessageStart]);
        if(Start > MB_N_C_0x){
          FC += 128;
          Exception = 2;
```

```
            break;
          }
          C[Start] = word(ByteReceiveArray[10 +
MessageStart],ByteReceiveArray[11 + MessageStart]) > 0;
#ifdef MbDebug
          Serial.print(" MB_FC_WRITE_COIL_0x C");
          Serial.print(Start);
          Serial.print("=");
          Serial.println(C[Start]);
#endif
          ByteReceiveArray[5 + MessageStart] = 6; //Number of bytes after this
one.
          MessageLength = 12;
          PopulateSendBuffer(&ByteReceiveArray[MessageStart],
MessageLength);
          Writes = 1 + Writes * (Writes < 999);
          FC = 0;
        }

        //**6***************** Write Single Register ******************
        else if(FC == 6)
        {
          Start = word(ByteReceiveArray[8 + MessageStart],ByteReceiveArray[9 +
MessageStart]);
          if(Start > MB_N_HR_4x){
            FC += 128;
            Exception = 2;
            break;
          }
          R[Start] = word(ByteReceiveArray[10 +
MessageStart],ByteReceiveArray[11 + MessageStart]);
```

```
#ifdef MbDebug
        Serial.print(" MB_FC_WRITE_REGISTER_4x R");
        Serial.print(Start);
        Serial.print("=");
        Serial.println(R[Start]);
#endif
        ByteReceiveArray[5 + MessageStart] = 6; //Number of bytes after this
one.
        MessageLength = 12;
        PopulateSendBuffer(&ByteReceiveArray[MessageStart],
MessageLength);
        Writes = 1 + Writes * (Writes < 999);
        FC = 0;
      }


      //**15********** Write Multiple Coils **********
      else if(FC == 15)
      {
        Start = word(ByteReceiveArray[8 + MessageStart],ByteReceiveArray[9 +
MessageStart]);
        CoilDataLength = word(ByteReceiveArray[10 +
MessageStart],ByteReceiveArray[11 + MessageStart]);
        if((Start+CoilDataLength) > MB_N_C_0x){
          FC += 128;
          Exception = 2;
          break;
        }
        ByteDataLength = CoilDataLength / 8;
        if(ByteDataLength * 8 < CoilDataLength) ByteDataLength++;
        CoilDataLength = ByteDataLength * 8;
```

```
#ifdef MbDebug
        Serial.print(" MB_FC_WRITE_MULTIPLE_COILS_0x S=");
        Serial.print(Start);
        Serial.print(" L=");
        Serial.println(CoilDataLength);
#endif
        buffer_save();
        ByteReceiveArray[5 + MessageStart] = 6; //Number of bytes after this
one.
        for(int i = 0; i < ByteDataLength ; i++)
        {
           for(int j = 0; j < 8; j++)
           {
              C[Start + i * 8 + j] = bitRead( ByteReceiveArray[13 + i +
MessageStart], j);
           }
        }
        MessageLength = 12;
        PopulateSendBuffer(&ByteReceiveArray[MessageStart],
MessageLength);
        Writes = 1 + Writes * (Writes < 999);
        FC = 0;
        buffer_restore();
     }


     //**16********** Write Multiple Registers **********
     else if(FC == 16)
     {
        Start = word(ByteReceiveArray[8 + MessageStart],ByteReceiveArray[9 +
MessageStart]);
```

```
                    if (Start == 256) {
                        Start = 0;
                        } else if (Start == 768) {
                            Start = 100;
                            } else if (Start == 12288) {
                                Start = 200;
                                } else if (Start == 12289) {
                                    Start = 202;
                                    } else if (Start == 12290) {
                                        Start = 204;
                                        } else if (Start ==
12291) {
                                            Start = 206;
                                            } else if (Start
== 12292) {
                                                Start =
208;
                                                } else
if (Start == 12293) {
        Start = 210;
        } else if (Start == 12294) {
        Start = 212;
        } else if (Start == 12295) {
            Start = 214;
            } else if (Start == 12296) {
                Start = 216;
```

```
                } else if (Start == 12297) {

                    Start = 218;

                    } else if (Start == 12416) {

                        Start = 220;

                        }
        WordDataLength = word(ByteReceiveArray[10 +
MessageStart],ByteReceiveArray[11 + MessageStart]);
        if((Start+WordDataLength) > MB_N_HR_4x){
          FC += 128;
          Exception = 2;
          break;
        }
        ByteDataLength = WordDataLength * 2;
#ifdef MbDebug
        Serial.print(" MB_FC_WRITE_MULTIPLE_REGISTERS_4x S=");
        Serial.print(Start);
        Serial.print(" L=");
        Serial.println(WordDataLength);
#endif
        buffer_save();
        ByteReceiveArray[5 + MessageStart] = 6; //Number of bytes after this
one.
        for(int i = 0; i < WordDataLength; i++)
        {
          R[Start + i] =  word(ByteReceiveArray[ 13 + i * 2 +
MessageStart],ByteReceiveArray[14 + i * 2 + MessageStart]);
        }
```

```
        MessageLength = 12;
        PopulateSendBuffer(&ByteReceiveArray[MessageStart],
MessageLength);
        Writes = 1 + Writes * (Writes < 999);
        FC = 0;
        buffer_restore();
        }

    //**80h+FC***************** Exception Handling ******************
    //    Jan 2014 - Andrew Frahn / Emmertex / emmertex@gmail.com          *
    //    Fix ported in from: https://github.com/emmertex/Modbus-Library   *
//*****************************************************************
    if (FC > 128) {
        //80h + FC = Exception
        buffer_save();
        ByteReceiveArray[7 + MessageStart] += 128;      //Turn FC echo into
Exception
        ByteReceiveArray[5 + MessageStart] = 3;         //Number of bytes after
this one
        ByteReceiveArray[8 + MessageStart] = Exception; //Exception Code
        PopulateSendBuffer(&ByteReceiveArray[MessageStart],9);
        FC = 0; //MB_FC_NONE;
        buffer_restore();
    }

    if (JustReceivedOne) {
        int i;
        MessageStart = MessageStart + 6 + ByteReceiveArray[5 + MessageStart];
#ifdef MbDebug
        Serial.print("/n Next start = ");
```

```
        Serial.println(MessageStart);
#endif
        if (MessageStart+5<TotalMessageLength) SetFC(ByteReceiveArray[7 +
MessageStart]);
        else {
          JustReceivedOne = false;
          FC = 0; //MB_FC_NONE;

#ifdef MbDebug
          for (i=0; i<NoOfBytesToSend; i++) {
            Serial.print(ByteSendArray[i],HEX);
            Serial.print(" ");
          }
#endif
          client.write(ByteSendArray,NoOfBytesToSend);
#ifdef MbDebug
          Serial.println(" sent");
#endif
          NoOfBytesToSend = 0;
          MessageStart = 0;
        }
#ifdef MbDebug
        Serial.print("TotalMessageLength = ");
        Serial.println(TotalMessageLength);
        Serial.print(" MessageStart = ");
        Serial.print(MessageStart);
        Serial.print(" FC = ");
        Serial.println(FC);
```

```
#endif
        }

    }

#ifdef MbRunsDebug
    Serial.print("Mb runs: ");
    Serial.print(Runs);
    Serial.print("  reads: ");
    Serial.print(Reads);
    Serial.print("  writes: ");
    Serial.print(Writes);
    Serial.println();
#endif

}

void Mudbus::buffer_save()
{
    int i;
    i=0;
    while(i<160)
    {
        SaveArray[i] = ByteReceiveArray[i];
        i++;
    }
}
void Mudbus::buffer_restore()
{
```

```
    int i;
    i=0;
    while(i<160)
    {
        ByteReceiveArray[i] = SaveArray[i];
        i++;
    }
}


void Mudbus::PopulateSendBuffer(uint8_t *SendBuffer, int NoOfBytes)
{
    int i;
    i=0;
    while(i<NoOfBytes)
    {
        ByteSendArray[NoOfBytesToSend] = SendBuffer[i];
        NoOfBytesToSend++;
        i++;

    }
}
void Mudbus::SetFC(int fc)
{
// Read coils (FC 1) 0x
    if(fc == 1) FC = 1;

// Read input discretes (FC 2) 1x
    else if(fc == 2) FC = 2;
```

```
// Read multiple registers (FC 3) 4x
    else if(fc == 3) FC = 3;

// Read input registers (FC 4) 3x
    else if(fc == 4) FC = 4;

// Write coil (FC 5) 0x
    else if(fc == 5) FC = 5;

// Write single register (FC 6) 4x
    else if(fc == 6) FC = 6;

// Read exception status (FC 7) we skip this one

// Force multiple coils (FC 15) 0x
    else if(fc == 15) FC = 15;

// Write multiple registers (FC 16) 4x
    else if(fc == 16) FC = 16;

// Read general reference (FC 20)  we skip this one

// Write general reference (FC 21)  we skip this one

// Mask write register (FC 22)  we skip this one

// Read/write registers (FC 23)  we skip this one
```

```
// Read FIFO queue (FC 24)  we skip this one
   else {
      Exception = 1;
      Serial.print(" FC Not Supported: FC=");
      Serial.print(fc);
      Serial.println();
      FC = fc + 128;

   }
}
```

A-1.5 Flasher_modified.h

```
#ifndef FLASHER_H
#define FLASHER_H

#include <Arduino.h>

class Flasher
{
   int greenLEDPin;
   int redLEDPin;
   long onTime = 200;
   long offTime = 200;
   unsigned long previousMillis;
         short _type; // 0 = non-parallel Bicolor LED's
                                 // 1 = parallel Bicolor LED's

  public:
```

```
    Flasher();
        void setPins(int theRedLEDPin, int theGreenLEDPin);
        void setBlink(long on, long off);
    void FlashRed();
    void FlashGreen();
    void FlashRedGreen();
    void Off();
    void Red();
    void Green();
        void setType(short type);
};

#endif
```

A-1.6 Flasher_modified.cpp

```
#include "Flasher_modified.h"

//class Flasher
//{
//    int greenLEDPin;
//    int redLEDPin;
//    long onTime = 200;
//    long offTime = 200;

//    unsigned long previousMillis;

//  public:
    Flasher::Flasher()
    {
    }
```

```
        void Flasher::setPins(int theRedLEDPin, int theGreenLEDPin)
        {
          redLEDPin = theRedLEDPin;
    greenLEDPin = theGreenLEDPin;
    pinMode(redLEDPin, OUTPUT);
    pinMode(greenLEDPin, OUTPUT);
        }

        void Flasher::setBlink(long on, long off)
        {
          onTime = on;
    offTime = off;
    previousMillis = 0;
        }

  void Flasher::FlashRed()
  {
          unsigned long currentMillis = millis();
          if (currentMillis - previousMillis >= onTime)
          {
                  previousMillis = currentMillis;  // Remember the time
                  if(_type == 0){
                          digitalWrite(redLEDPin, !digitalRead(redLEDPin));
                          digitalWrite(greenLEDPin, HIGH);
                  }
                  else{
                          digitalWrite(greenLEDPin, HIGH);
                          digitalWrite(redLEDPin, !digitalRead(redLEDPin));
```

```
                }
                //digitalWrite(greenLEDPin, LOW);
        }
}

void Flasher::FlashGreen()
{
  unsigned long currentMillis = millis();
  if (currentMillis - previousMillis >= onTime)
  {
    previousMillis = currentMillis;  // Remember the time
                if(_type == 0){
                   digitalWrite(greenLEDPin, !digitalRead(greenLEDPin));
                        digitalWrite(redLEDPin, HIGH);
                }
                else{
                        digitalWrite(greenLEDPin, !digitalRead(greenLEDPin));
                        digitalWrite(redLEDPin, HIGH);
                }
    //digitalWrite(redLEDPin, LOW);
  }
}

void Flasher::FlashRedGreen()
{
  unsigned long currentMillis = millis();
  if (currentMillis - previousMillis >= onTime)
  {
    previousMillis = currentMillis;  // Remember the time
```

```
    digitalWrite(greenLEDPin, !digitalRead(greenLEDPin));
    digitalWrite(redLEDPin, !digitalRead(greenLEDPin));
   }
  }

  void Flasher::Off()
  {
   digitalWrite(greenLEDPin, HIGH);
   digitalWrite(redLEDPin, HIGH);
  }

  void Flasher::Red()
  {
            digitalWrite(greenLEDPin, HIGH);
            digitalWrite(redLEDPin, LOW);
  }
  void Flasher::Green()
  {
   digitalWrite(greenLEDPin, LOW);
   digitalWrite(redLEDPin, HIGH);
  }

      void Flasher::setType(short type){
            _type = type;
      }
//};
```

List of Symbols, Abbreviations, and Acronyms

A/D	analog/digital
EEPROM	Electrically Erasable Programmable Read-Only Memory
I^2C	inter-integrated circuit
IC	integrated circuit
IDE	integrated development environment
I/O	input/output
IP	Internet Protocol
ISP	In-System Programming
LED	light-emitting diode
MBE	molecular beam epitaxy
PC	personal computer
PWM	Pulse-Width Modulation
RISC	Reduced Instruction Set Computer
SPI	Serial Peripheral Interface
SRAM	Static Random-Access Memory
SSR	solid-state relay
TCP	Transmission Control Protocol
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus

1 (PDF) DEFENSE TECHNICAL INFORMATION CTR DTIC OCA

2 (PDF) DIR ARL
IMAL HRA
RECORDS MGMT
RDRL DCL
TECH LIB

1 (PDF) GOVT PRINTG OFC
A MALHOTRA

1 (PDF) DIR ARL
RDRL SEE I
R ENCK